68000 PROGRAMMER'S HAND BOOK

プログラマーズ・ハンドブック　宍倉幸則 著

技術評論社

68000 PROGRAMMER'S HAND BOOK

プログラマーズ・ハンドブック　宍倉幸則 著

技術評論社

68000 PROGRAMMER'S HAND BOOK

プログラマーズ・ハンドブック

宍倉幸則 著

技術評論社

〔序〕

筆者が初めて68000のドキュメントを手にしたのは，1979年か80年のマイクロコンピュータ・ショーであったように記憶しています．

現在はみかけなくなりましたが，当時は各社ブースの他にマニュアルセンターが用意され，様々な技術文献が陳列されていました．その中に直輸入版の〝MC68000 16-BIT MICRO PROCESSOR User's Manual〟なるものがあり，すでに68000の概要は雑誌などで紹介されていましたので，早速求めることにしました．

表紙には〝Break away from the past〟と記され，実に印象的でした．マニュアルを読み進めていくにつれ，次のような感想を持ちました．

「この石には夢がいっぱい詰まっている」と……．

あれから七年近い歳月が流れ，何十万円もしていた68000は８Ｍバージョンのセラミック・パッケージで3000円となり，何千円もしていた16ＫビットD-RAMは，現在では256Ｋビット素子が量産され価格も500円前後となりました（価格はいずれも'86年初旬の秋葉原価格）．この結果，数年前には考えられなかったことですが，個人でさえ68000に１Ｍとか２Ｍバイトのメモリを搭載したシステムも設計可能であり，未来の夢の68000は我々の68000となりました．

68000は進化したアーキテクチャをもっています．このことは，マイクロプロセッサの未来そのものであり，誰が設計しても似たようなアーキテクチャに行き着く地点を暗示しています．もちろんさらに進化するでしょうが，少なくとも，辿らざるを得ないアーキテクチャを先取りしたプロセッサであることに異論を唱える者はいないでしょう．

本書は68000というマイクロプロセッサが，仕事にせよ趣味にせよ，「自分にとって必要である」と思われている方々への十分な技術資料となり得ることを確信しています．

このような見地から，道理にかなってはいるが，実際には役に立たないと判断されるような事柄をもふまえ，洗練されたアーキテクチャをもつ68000というプロセッサのアセンブリ言語によるプログラミングが，いかに容易で楽しいものであるかを，ひとりでも多くの方々に知っていただき，68000にたずさわる技術者の座右の書として，毎日どこかのページが参照されることを願っています．

本書の執筆を薦めて下さった技術評論社第一編集部部長・幡垣氏と編集スタッフの皆様に感謝の意を表します．

なお本書の執筆にあたり，株式会社ライフボートより，MS－DOS上で利用できる68000/68010/68020の開発ツールとして，Quelo，ならびにμ-Series Assemblers，２点のクロスアセンブラを拝借いたしました．使用する機会を与えて下さった関係者の方々に，誌面を借りてお礼申し上げます．

1986年８月　　宍倉幸則

■本書を利用するにあたって

本書は，アセンブラでは大きなプログラム開発は不可能だ，アセンブラは難解で，といったアセンブリ言語の神話を覆し，これらの神話が68000に関しては適切でないことを理解していただき，68000というプロセッサ上で，正しく動作する実践的なプログラム開発が可能になるように，「アセンブリ言語をツールとして使う際の体系的な考え方」を理解していただくことにありますが，プログラム開発の際の技術資料となり得るようにも十分な配慮がなされています．

プログラミングは，ある意味では〝慣れ〟ですから，熱意のある読者であれば，比較的短期間である程度の成果を期待できるかも知れません．だからと言って，ニーモニックの詳細まで記憶しようとするのは時間の無駄であり，それだけに命令を体系的に整理し，必要な時には即座に取り出せるようにしておけば，こんな便利なことはありません．

本書では特に以下のように配慮致しました．

① 形式的な一般論ではなく，〝ホンネ〟を言うことにより，実践的かつ視野の広い事柄に言及し，現場での混乱に対処できるような〝免疫〟となり得るよう解説を心掛けたこと．
② 適切な項目に分類し，各項目の先頭には必ず要点をまとめ，読者の興味意欲をそそるように心掛けたこと．
③ 特に重要な必修事項に関しては，特別に紙面を割り当てて詳細な解説を行い，読者自身が時間を浪費しないように心掛けたこと．
たとえば，「68000のアドレッシングは何々です」ではなく，「アドレッシングとは何なのか」という解説をし，理解の一助となるようにしたこと．
④ ニーモニックを実践的観点から整理し，そのページを開けば，すべての情報が把握できるようにしたこと．
⑤ サブルーチンに関しては，単なる例題ではなく，それだけで部品として即使用できるような汎用性のあるものとし，入力／出力条件を解説内に明記したこと．
⑥ ハードウエアに大きく依存する割り込み処理などに関しては，信号を１本１本たどりながら解説しなければ説得力に欠けるものと判断し，解説だけに留め，割り込み処理プログラムを割愛したこと．

目　次

第2部 プログラム …… 123

第3部 命令の詳細 265

第1部　基礎知識

本セクションでは,68000というプロセッサのもつアーキテクチャのすばらしさと,開発言語としてのアセンブラについて，結論的には,「68000を使いましょう」いや「使うべきだ」という根拠について述べられています.

◎68000とアセンブラ

1 なぜ68000か

当然のことながら，コンピュータには記憶空間が必要であり，より複雑な作業をするには，それなりのプログラムを格納するためのメモリが必要です．つい最近までは信じられなかったことですが，広大なメモリ空間も今となっては現実であり，その意味では自由に行動範囲を拡張できるようになりました．もちろん，小規模なシステムは無意味であるというのではありません．

ここでは，メモリ空間のもたらす意味と68000の管理できるメモリ空間，アーキテクチャ，そのアーキテクチャのもたらすプログラミング環境について述べます．

［1］メモリ空間

8ビットプロセッサとして現在も量産され多方面で活躍しているZ－80などと，16ビットプロセッサとの最大の相違点は，アクセスできるメモリ空間といえましょう（ちなみに，Z－80では64Kバイト，PC－9801など多くの16ビットパソコンに搭載されている8086では1Mバイト）．

ご承知のように，コンピュータには記憶素子が必要であり，これがなければどうすることもできません．

外部記憶装置としてのフロッピーディスクやハードディスクによって，数Mバイトとか20～50Mバイト程度の仮想メモリを考えることができますが，命令そのものはメモリに対するものであり，一度はメモリ上へ読み込まねばなりません．

マイクロプロセッサ自身が管理できるメモリ空間には，主としてプロセッサ自身が実行すべき命令が格納され，我々にとってはより有意義な，コンピュータにとっては複雑な作業をしてもらうには，どうしても広いメモリ空間が必要なのです．8ビットプロセッサには〝8ビットのよさ〟があることは認めますが，PC－8801でワープロを使うのとPC－9801シリーズで使うのとは〝雲泥の差があり〟，この最大の要因はメモリ空間なのです．

［2］68000の管理できるメモリ空間

68000は16Mバイトのメモリ空間を管理することができますが，実際には，スーパーバイザとユーザ空間の2つのメモリ区分を持ち，さらに各区分はプログラム領域とデータ領域に区分け可能ですから，4つのメモリ空間が存在します．どの領域であるかのステータスは，FC（ファンクション・コード）としてプロセッサから出力され，これらを，メモリの選択に使用すれば，64Mバイトまでのメモリ領域を管理できます（ただしメモリ空間をどのようにマップするかは，ハードウェアの設計エンジニアに委ねられる）．

小量のメモリでも68000は十分すばらしい仕事をしてくれます．しかし現実には，「もっとメモリが欲しい」というのが正直なところであり，メモリ素子が安価であり容易に入手可能になった今こそ，68000の登場となるわけです．

8086のプログラミングを経験された方は承知のことと思いますが，8086のアドレスレジスタはZ－80と同様16ビットであり，64Kバイトしか管理できません．それ以上はセグメントレジスタをプログラマ自身が設定しなければならず，非常な重荷を背負わせられている

のが現状です．具体的にどのように困るのかは実例をあげねばなりませんし，本書の領域でもありませんから，これ以上申しあげないことにします．

大切なことは，8086のデコボコしたメモリ空間に対し，68000のそれはフラットなものであるということです．

［3］68000のアーキテクチャのもたらすプログラミング環境

68000はアナウンスされた時点から筆者の恋人でした．これはマイクロ·エレクトロニクスに従事される多くのエンジニアにしても，同感ではないかと思われるし，チップの設計開発エンジニアであれば，「どうして我々にこのような発想が生まれ得なかったのだろうか」と思う一方で，「やられた」という感慨をいだかれたかも知れません．

68000のプログラミング環境がいかに優れたものであるかを，具体的に説明するのは容易ではありません．実際にプログラミングしてこそ，換言すれば，68000を使ってこそ理解できるものですが，少なくとも，Z－80や8086のアセンブラで重くのしかかっていた制約は，68000ではまったく存在しません．アセンブリ言語の場合は，特にマシンのアーキテクチャをそのまま反映したものになりますから，アーキテクチャが洗練されたものであれば，それだけアセンブラでのプログラミングも容易であることを申し上げておきます．この意味で，アセンブラでつまずいた先輩のアドバイスである「アセンブラは難解だ」という表現も，場合によっては適切でないこともあり得ることを忘れないでください．

計算機に要求されるべきアーキテクチャには，コンピュータがプログラムで走る以上，ある一定の定石があることに気がつきます．この意味で68000は，ストアード·プログラム方式（第五世代コンピュータに対する表現）の16ビット・マイクロプロセッサとしては，〝究極〟なのであり，32ビットのアーキテクチャはそのまま68020へ受け継がれています．

それではソフト面から68000の魅力について簡単に整理してみようと思います．

プログラマからは32ビットのプロセッサであること

まず，68000の内部が32ビットであることにふれなければなりません．プログラマにとって32ビットのプロセッサであることが，16ビットプロセッサとどう違うのか，そのメリットについては述べるまでもないことです．

徹底的に汎用化された豊富な内部レジスタ

レジスタが汎用化されているということは，それだけプログラムステップが少なくてすむことを意味し，特定のレジスタしか許されていなければ，そのレジスタを使うにはどこかに退避し，再び復帰するなどの命令を記述することになり，しかも，このような操作は頻繁に要求されるので，プログラマに選択できる余地は極めて限られたものとなります．

プログラミング過程では注目すべきデータは複数であり，ちょっとしたプログラムでさえ数個とか10個となり，より複雑な処理になればなるほど増大します．しかも，これらは互いに密接な関係を持っているはずであり，使うレジスタ（使いたいレジスタ）が特定のものでそのレジスタに何か意味のあるデータが格納されていれば，他のレジスタでは代用できないのですから，次の処理で使うために退避／復帰を余儀なくされます．

たとえば，保持すべきアドレスポインタが３つとか４つあり，一度はポインタを設定したものの，他のポインタをセットしたいために書き換えなければならないが，この値を失っては処理を続行できない，などということは頻繁に要求されます（この程度のことは通

常のプログラム開発では常識である)．結局，「プログラミングの途中でデータがどのようになっているかを見失ってはならない」ということで，プログラマはいくつものチェックポイントを通過しなければなりません．68000では，以上のようなチェックポイント(制約)は非常に少なくてすみますから，実にすっきりとした記述が可能なのです．

強力なアドレッシング

アドレッシングモードのセクションで詳細な解説がなされていますが，簡単にふれておきます．

アドレッシングとはデータへのアクセス方式を意味しますが，一方，プログラミングとは，「メモリやプロセッサ内のレジスタなどを介したデータの加工」なのですから，この指定方式が強力であればあるほどプログラミング環境は充実したものとなり，プログラマへの負担は大幅に軽減されることになります．

強力な命令セット

高度で洗練された命令セットは，Cコンパイラなどの高級言語などを実に効率よくサポートするものですが，アセンブラでプログラミングをする際にも実にすっきりとした記述が可能になり、極めて見通しのよいプログラムを書くことができます．

2 なぜアセンブラか

アセンブラを使うメリットについて述べるには，アセンブラの意味を明確にしておかねばなりません．ここでは，耳慣れた言葉である〝機械語〟と機械語を生成するアセンブラと呼ばれるコンピュータ（プログラミング）言語の説明をし，さらにアセンブリ言語の特長や問題点などについて整理しています．

[1] コンピュータ言語としての機械語

どのようなコンピュータでもその内部はディジタル回路ですから，電圧が高いか低いかで動作しており，電圧の高い状態を〝1〟，低い状態を〝0〟に対応させると，コンピュータ（プロセッサ）は〝1〟か〝0〟かで動作していると説明できます．〝1〟か〝0〟かということは，すなわち2進であり，我々の意志は2進数としてプロセッサへ伝えられねばなりません．

高級言語では文を記述するようにしてプログラミングできますが，この文は複数の機械語に変換されてプロセッサへ伝えられており，我々がアセンブラでプログラムを記述しようが，CコンパイラやBASICで記述しようが，プロセッサへは2進コードとして与えられていることになります．この意味で，コンピュータ言語とはマシン語（2進コード）変換プログラムであるわけです．

[2] アセンブラ

MOVE.W D0,D1という命令は，MPU内のD0（データレジスタ0）の下位16ビットをD1（データ・レジスタ1）へ転送（コピー）する命令ですが，プロセッサ自身には「0011001000000000」として与えなければなりません．〝1〟は高い電圧で〝0〟は低い電圧ですが，通常は16進数で表記し，MOVE.W D0,D1のマシンコード(オブジェクトコード)は，＄3200であると言います．

＄3200では人間に理解しにくいので，コンピュータの命令セットの1つ1つにニーモニック(略語)を割り当て，たとえば，MOVE.W D0,D1と記述すると，＄3200という機械語に変換するのがアセンブラと呼ばれる言語です。

このようにアセンブラに受け入れられる記述は略語であり，コンピュータの命令セットそのものであるので，Cコンパイラなどの高級言語での〝printf〟のような，より具体的な表現は許されません．どのようなプロセッサでも同じですが，たとえば68000には〝printf〟という命令を理解できないので，コンパイラは，68000が実行可能な命令セットのいくつかを組み合わせて，〝printf〟という文を機械語に変換（コンパイル）します．

アセンブラが理解しにくい理由は，コンピュータの命令セットそのものを記述しなければならず(ニーモニックはコンピュータの命令と1対1で対応する)，先ほどのMOVE命令が〝printf〟にとって，どのような意味を持つものか把握できないことにあり，プログラマ自身が処理全体をイメージできないという点にあります．しかし決して難問ではなく，時間の経過と共に各命令の輪郭が理解できるようになりますから，最初に少々理解できないことがあっても気にしないことです．

1つの命令では何もできませんが，これらを複雑に組み合わせることによって，すばらしい成果が期待できることを申し上げておきます．

[3] アセンブラによるプログラム開発のメリット

プログラミング言語を何にすべきかという選択をせまられる時，プログラマの置かれている立場によって意見が分かれるものと思われます．これは専業プログラマに限定したことではなく，プログラム開発に明けくれるすべての方々を対象としたものです．

「必要としない」とは，要求されないだけでなく，現状に不便も感じていないということで，多くは，与えられた環境から脱皮する必要のないプログラマということになるでしょうか．アセンブラ経験者の多くは他人から薦められて使い始めたわけではなく，「自分に必要である」という強い意志からアセンブラをマスタしたのだと思います．

以下はアセンブラを使うことによるメリットを筆者なりの〝ホンネ〟で整理したものであり，アセンブラを使えなくとも仕事はできるが，自然淘汰したくなければアセンブラを使えるようにしておくべきであり(使うべきである)，最後の切り札としてのアセンブラに精通しておいて損はない，といった内容になっています．

開発言語(ツール)入手の現状から

誰もが使っているポピュラーなMPU (CPU) であれば，外部から提供される開発ツールが豊富ですが，コンパイラともなると，まともな開発環境を獲得するには，16ビットパソコン本体程度の出費を覚悟しなければなりません．しかも，使いたいと思っている石（プロセッサ）のコンパイラが即座に入手できるわけでもありません．

開発元では，商売になり得るプロセッサ用のコンパイラを商品化するわけですから，一般的な結論として，高級言語で開発をしたいと思った時点では，商品化されたツールは存在しないというのが現状です．今でこそMS−DOS上で走るCコンパイラが豊富ですが，インテルの8086が個人レベルで十分無理のないコストになった時期でさえ，Cコンパイラの使用など夢であったわけです．

ツールの信頼性

たとえばCコンパイラなどの高級言語で開発を経験された方であれば，そのコンパイラに慣れるまでは大変であり，雑誌や書籍でのCコンパイラの評価と現実とのギャップに失望されたかもしれません．コンパイラにしてもリンカにしても，ソフトウエアである以上バグが皆無ということは考えられず，このようなことを承知で商売せざるを得ないのが現状です．

一般にはコンパイラが出荷されてから2～3年を経ないと〝マトモ〟にならないと言われ，その間はこのような制約下で作業せざるを得ません．筆者の経験では，アセンブラ(機械語へ変換するプログラムとしてのアセンブラ）に関しては重大なバグを経験していません．

アセンブラを使えると，ひとり歩きができる

趣味であれ仕事であれ，使いたいプロセッサのソフトウエアを開発するのに、ツールがなければどうすることもできません．この場合の強力な手段としてクロスアセンブラがあり，たとえば，CP/M−80上で68000のオブジェクトコードを生成するプログラム（これを

クロスアセンブラという）を開発し，これでアセンブルしてターゲットへロードするわけです．クロスアセンブラが市販されていればそれを購入することもあるでしょうが，いずれにせよ，機器組み込み用プログラムの殆どはこのようにして開発されたはずです．

これに対して，コンパイラにせよBASICにせよ，与えられた環境でしかプログラミングできないのであれば，一歩も開発を進めることはできません．たとえばBASICでなら数行で記述できるような処理であっても，68000ボード上にはBASICの文法を解釈するプログラムがなければならないわけですから，この程度のレベルの仕事ができる人が少数であるのも当然といえましょう．

ハードウエアとの兼ね合い

PC－9801シリーズなどのパソコン上でプログラミングしていると，つい忘れてしまうことなのですが，このマシンを開発したエンジニアも時には思い通りの結果が得られず，その原因究明に我を忘れたことでしょう．メモリは正しく作動しているだろうか，フロッピーディスクのインターフェースは完全であろうか，等々です．

ハードウエア開発の最前線にあっては，テストプログラム自体をあらかじめ作成しておくわけですが，せいぜい0.5Kバイト程度のプログラムサイズで十分ですし，内容はその場で即座に修正できなければなりません．また単一マシンサイクルの実行も要求されます．この点でコンパイラの出力する機械語では複数の機械語が走ってしまい，何をテストしているのか不明になってしまいますから，結局原因の究明は極めて困難となります．

高速なプログラムの作成

アセンブラで作成したプログラムが，最もアセンブラに近い開発言語であるCコンパイラより高速であるのは当然なのですが，その理由を少し考えてみることにしましょう．

Cコンパイラはアセンブラと同様なオブジェクトを生成できますが，100％アセンブラで記述したような，最適化されたオブジェクトが得られるわけではありません．すでにおわかりかと思いますが，コンパイラは記述された命令（キーワード）を複数の機械語に変換せざるを得ませんから，ある特定の命令はアセンブラで記述したように無駄なく変換できても，最適化には限界があるわけです．

したがって，１つの処理をするために最適化された命令を実行するのと，これに対して複数の命令で対処するのとは，オブジェクトサイズに大きな相違が発生します．ちなみに，Cコンパイラのオブジェクトサイズは，アセンブラの５～10倍程度といわれています．

これは単なる一例にすぎませんが，Cコンパイラに限らず高級言語では変数を使用でき，この変数がプログラミングを著しく容易にします．しかもCコンパイラの場合は変数を高速にアクセスできるような構造を持った言語なのですが，変数はメモリ領域へ割り当てなければならず，複雑なメモリアドレスの計算をしてからでないと変数を操作できません．ですから，Cコンパイラの出力する機械語には，変数操作のために必要な部分がどうしても残ってしまいます．

一方，アセンブラでは操作しようとする変数のアドレスは，プログラミング段階でMPU内のアドレスレジスタへ直接プログラマがセットしており，いきなり変数のアクセスを開始できるわけです．コンピュータの仕事はメモリの操作といってもよいでしょうから，これが処理スピードの違いとなってしまうわけです．

プログラムそのものがプロセッサの動作のすべてであるということ

先ほどアセンブラのニーモニックがコンピュータの命令セットと1対1で対応していることを述べましたが，これは極めて興味深い意味あいを含んでいます．

コンパイラの出力するオブジェクトコード（マシンコード）を解析すれば明確なことですが，ある一定の処理系に従ってマシン語へコンパイルされるのですから，長期間このような作業をすれば，ある程度は，どのような機械語へ変換されるのかを想像することも不可能ではないでしょう．しかし，プログラム全体がどう変換されるかを解析するのも大変ですし，そのためにはコンパイラの内部構造に精通していなければなりません．

一般にコンパイラによって出力される機械語がどのようなものであるか，プログラマにはまったく不明です．これに対してアセンブラの場合は，マシン（プロセッサ）の命令セットを記述するのですから，ある命令を実行する時のクロックサイクル数，アドレスバスやデータバスの内容，各種制御信号の状態，これらの時間的推移（タイミング）に至るまで把握することができます．

マシンの状態を把握できることは大切ですが，ハードウエア開発に必要であること以外に，原子炉とか医療機器あるいは宇宙船などのミスの許されない人命にかかわるような分野においては，エージングされた命令で記述せざるを得ないし，「今何をしているのか」「システムはどう応答しているのか」は，的確に把握されなければなりません．

このようなことから，マシンの基本単位（これ以上は分割できない命令）でプログラミングするのであれば，「予測できないトラブル」の可能性はゼロに収束するわけです．

［4］開発言語としてのアセンブリ言語とその適用分野

筆者はすべての開発にアセンブラを使用せよ，と言いたいのではなく，一通り使えるようにすべきであり，大きなプログラム中でのいくつかのモジュールは，アセンブラで記述すべきだと思います．それに最後に頼りになるのはアセンブラしかなく，何か困った時に決して無駄になるようなことはありません．

大きなプログラムをアセンブラで記述するには無駄が多く，採算的に問題が多いといわれますが，これはひとりで開発する場合の話であり，1カ月あたり数Kバイト程度の開発は十分可能ですから，内容にもよりますが，4～5人のスタッフなら相当大きなプログラム開発も容易です．

以下はこれまでのまとめになりますが，アセンブラで開発すべき典型的な分野を整理してみました．

① ハードウエアの動作を確認するためのデバッグソフト
② ROM上に常駐すべきプログラム
③ OSのBIOS（入出力を受け持つサブルーチンの集まり）
④ リアルタイム・モニタ
⑤ グラフィックスなどの特に高速性が要求される分野のプログラム
⑥ 機器組み込用プログラム（ロボット，FFTアナライザ）
⑦ セキュリティ・システム

[5] アセンブラが初心者にとって難解であると思われる一例

アセンブラでプログラミングする際の最初の壁は，「アドレス」という概念ではないでしょうか．つまり，「コンピュータとは何者なのか」ということが理解できなければならず，MPUとメモリとのインターフェースなどの設計ができなくても，これらの役割や動作に関する的確な知識が必要であることです．

変数Aと変数Bがあり，両者の和をCという変数に格納したい場合，BASICなら，

C＝A＋B

と記述しますが，アセンブラでは，A，B，Cなどの変数をメモリのどこへ予約するのかというのが先決ですし，変数を操作するには，そのアドレス（ポインタ）をアドレスレジスタへ設定しなければなりません．さらに，変数のサイズによって予約すべきメモリサイズも異なり，あたかも自分がメモリを操作しているような気持ちになりきらなければならないことです．

キーワードを説明するなら，次のようになるでしょう．

■メモリ　コンピュータには命令やデータを記憶する場所が必要であり，これをメモリという．そしてメモリへ記憶させたり，記憶された内容を取り出す時には，記憶させた同じ場所から取り出せなければならない．このために番地(アドレス)があり，メモリを操作するには，〈アドレス〉が必要である．

プロセッサ内には，アドレスを指定するための一時記憶メモリがあり，これをアドレスレジスタと呼び，アドレスのことを単にポインタと呼ぶこともある．

■データ　コンピュータがデータを操作すためにはデータ構造についての理解も必要であり，符号を考慮しない場合，1バイトでは0～255まで，2バイトなら0～65535までしか表現できず，処理に要求されるデータがどのようなものであるか把握しておかねばならない．

一般に，メモリはバイトマップ構成であるので，2バイトのデータを操作するのであれば，連続した2つの番地が割り当てられる．

さて，68000のアセンブラで，A，B，Cのデータサイズが各2バイトの単なる数値である場合，C＝A＋Bは次のように表現されます．

```
        LEA      VAL,A0 ···············A0に変数の先頭アドレスをセット
        CLR.W    D0 ···················D0を払う（ここに答を入れる）
        ADD.W    (A0)+,D0 ·············D0と変数Aを加算して結果をD0に
        ADD.W    (A0)+,D0 ·············D0と変数Bを加算して結果をD0に
        MOVE.W   D0,(A0) ··············変数AとBの和を変数Cへ格納
VAL     DS.W     1 ····················変数Aに2バイトの領域を予約
        DS.W     1 ····················変数Bに2バイトの領域を予約
        DS.W     1 ····················変数Cに2バイトの領域を予約
```

[6] どうすればアセンブラをツールとして活用できるようになるか

これは本書のメインテーマでもありますが,「こうしたらよいであろう」と思われる事柄を以下に整理してみました.

はじめから多くを望まず,わからないからといって失望したりしない.

これが最も大切なことであり,要するに「好きなようにやればよい」ということ.すべての命令セットを理解しなければプログラミングできないわけではなく,実践によって命令の働きを理解することが大切です.それゆえ,「考えること」は必要ですが,「憶えることは」要求されません.

プロセッサの働きの概要をイメージ(把握)する

アセンブラはハードウエア(マシン)に直接働きかけるプログラミング言語ですから,ハードウエアの設計ができなくてもよいが,プロセッサの働きの概要を理解しておくべきです.そうすれば,各命令によって「何が操作されるのか」ということを,鮮明にイメージすることが可能になります.

本書の例題を再編成し拡張してみる

命令のすべてを知らなければプログラミングできないわけではなく,思いつくことを実行してみる.

たとえば,1～10までの総和を求めるプログラムが例題にあれば,これで終わりにせず,それまでの知識を基礎にして違う角度からプログラムを編成してみる.

この例なら,1～10の総和を求める処理系を拡張し,対象となる数値の範囲を設定できるようにしてみるとか,偶数だけを加算するとか,とにかくいろいろやってみる.

興味のある分野の研究

たとえゲームであっても,人を魅了するような動きの画面を実現するのは容易ではないでしょう.あるいはデータベースに興味があれば,高速なデータアクセスを可能とする記憶方法など,それぞれの分野に必要なアルゴリズムの探求をすればよいのであり,とにかく,応用できる分野をターゲットにすることも大切です.

要するに,「無理やりマスターしょうとせずに楽しむ」ことも必要だし,楽しめる分野を開拓し,その道で商売できるようになれば,こんなすばらしいこともありません.

設計資料の作成

成功したプログラムの整理をし,これを次回への設計資料にすることも重要です.

3 レジスタ内のデータ構成

アセンブラでのプログラミングに入るには，まずレジスタやメモリ内のデーター構成を把握しておく必要があり，内部レジスタの役割や68000が扱えるデータについて理解しておく必要があります．

コンピュータには一度に扱える（処理できる）データに制約があり，16ビットのデータサイズであれば，0～65535までの65536通りの数値しか表現できないし，英数字なら2文字，漢字では1文字しか処理できません．しかしプロセッサの外部にメモリを実装することによって，同様な処理を繰り返し実行することができ，高精度演算や文字列が扱えるなど，我々に都合のよいような仕事をさせることが可能なのです．

68000が扱うことのできるデータサイズ，すなわちオペランドについての説明を前置きしますが，MPUレジスタやメモリ内のデータ構成の説明でも，レジスタ内のデータやメモリ内のデータは，すべて〝オペランド〟（操作対象）であることを把握しておいてください．

［1］オペランドのサイズ

実行する命令によって操作される内容（オペランド）にはサイズがあり，68000は次のようなオペランドをサポートします．

バイト	：8ビット
ワード	：16ビット
ロングワード	：32ビット

各命令のオペランド・サイズは，命令の中で指定するタイプ（エクスプリシット）と，命令により暗黙的に指定されるタイプ（インプリシット）とがあり，前者はバイト，ワード，ロングワードのすべてのオペランド・サイズをサポートし，後者は3つのオペランドサイズのうちの，あるサブセットをサポートします．

つまり，オペランド・サイズの指定を必要とする命令と，そうでない命令とに区別できるというだけのことです．

［2］MPU内のレジスタとその役割

68000内部には次のような名称のレジスタ（一時記憶メモリ）があり，アセンブラでのプログラミングでは，これらのレジスタの役割を理解することは必修事項です（図1.1）．

D0～D7	：8本の32ビット・データレジスタ
A0～A6	：7本の32ビット・アドレスレジスタ
SP	：2本の（SSPとUSP）32ビット・スタックポインタ（A7が使用される）
PC	：1本の32ビット・プログラムカウンタ
SR	：1本の16ビット・ステータスレジスタ

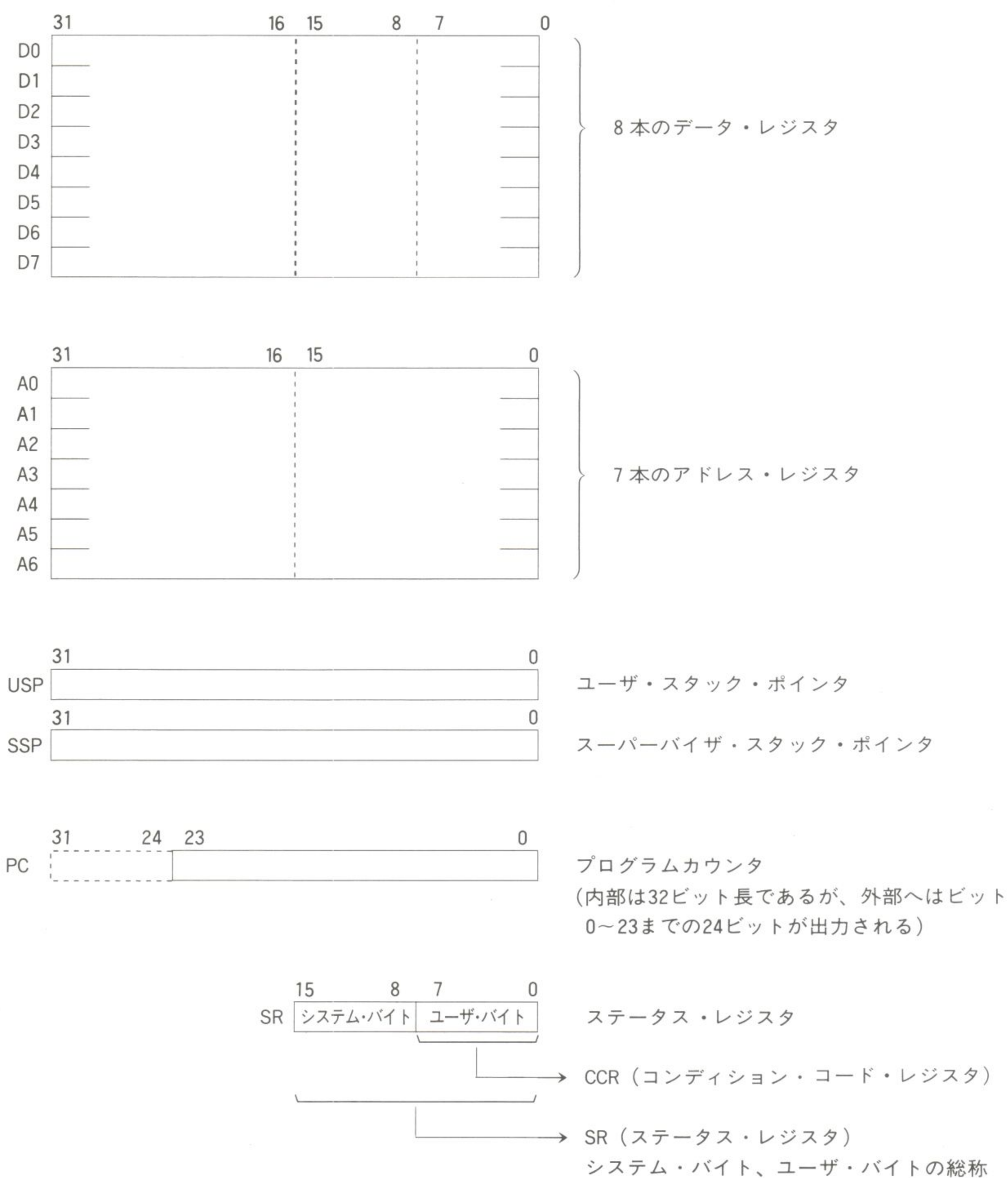
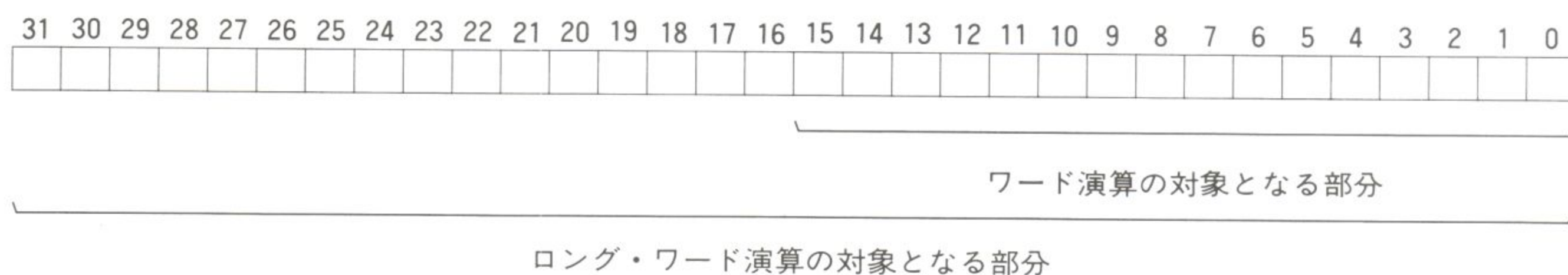
図1.1　MPU内のレジスタ構成

note：USP，SSPにはA7が割り当てられる

図1.2　アドレス・レジスタの構成

モトローラがアドレスレジスタとデータレジスタにその役割を分担したことを,「扱いにくい」と表現する人もあるようですが，アドレスレジスタをデータレジスタに，あるいはデータレジスタをアドレスレジスタに使う根拠もないように思えるし，本来〝アドレス〟と〝データ〟とは異質のものであり，むしろモトローラの選択は適切であったと，筆者は考えています．

アドレスレジスタにはアドレス(ポインタ)，データレジスタにはデータ以外の値は存在しませんから，ポインタと単なるデータとを混同するチャンスは激減します．アセンブラの経験者であれば，このようなプログラミング環境の恩恵を即座に評価できるはずです．きっと苦い経験をしているでしょうから…….

Cコンパイラではポインタが〝アドレス〟に相当しますが，単なるデータとポインタとの混同はアセンブラと同様に致命的であり，このようなバグは容易にとれない危険性があることからも，アドレスレジスタとデータレジスタが明確に区別されていることは，致命的なバグを抑制する大きな要因になっています．

アドレスレジスタ［A0～A7］図1.2

コンピュータにはメモリが必要であり,実行させようとする命令はもちろんのこと,様々なデータを記憶させる領域（メモリ）が必要です．そして，メモリへデータを格納したり取り出したりする際には，その場所を指定する必要があります．この場所のことを〝アドレス〟と呼んでいます．

68000の内部にはA０～A７までの８本のアドレスレジスタがあり,いずれもインデックスレジスタとしても使用可能ですが，後述のようにA７はシステム・スタックポインタに割り当てられています．

プロセッサがメモリをアクセスする時には,必ずMPUのアドレスバスからアドレスが出力されますが，この時のアドレスを保持するのがアドレスレジスタです．換言すれば，アドレスレジスタの内容がアドレスバスから出力されます．このようにプログラマは必要に応じてアドレスレジスタへアドレスを設定し，所定のメモリを操作することになります．

アドレスレジスタは32ビット長であり，2^{32}の異なった場所（アドレス）を指定できますが,アドレスバスはA０～A23までの24ビットがサポートされ，2^{24}の指定ができることから，16Mバイト（16,777,216）の記憶場所を操作することができます．ただし，アドレスバスのA０は上位／下位バイトを区別するためにMPU内部で使用され,実際のアドレスバスはA１～A23までの23ビットとなっています．

【アドレスレジスタとオペランド・サイズ】

アドレスレジスタに対する操作（広い意味での演算）は，ワードやロングワードのサイズが許され，バイト演算は禁止されます．

1．ワードサイズとアドレスレジスタ

ワード演算では下位２バイトの16ビットが操作の対象になりますが,ソース･オペランドに指定されたアドレスレジスタに対し，MPU はアドレスレジスタの下位半分を読み出して演算するだけでなく，ビット15の内容をビット16～31にコピーし，符号拡張した32ビットの値として演算します．

このため，ビット15がゼロ〝0〟ならビット16～31はゼロ〝0〟で満たされ，結果的に予定通りの演算結果となりますが，イチ〝1〟であれば，ビット16～31は〝1〟で満たさ

れ，オリジナルの16ビットデータとは異なった値を内部で生成して演算します．

たとえば，＄7FFFと＄8000が符号拡張されると，

＄7FFF ——→ ＄00007FFF

＄8000 ——→ ＄FFFF8000

となります．

ちょっと余談かもしれませんが，現実問題として，アドレスが16ビットとして使用されることがあるのかどうか，考えてみました．

16ビットのアドレスで指定できるメモリ空間は64Kバイトですし，現在のメモリ素子の集積度から考えても，実装メモリ空間が64Kということは非現実的です．そこで，我々がプログラミングする場合には，32ビットのアドレスを指定しなければならないでしょうから，アドレスレジスタに対するワード演算が要求されることは少ないと思われます．

メモリ空間が64Kバイト以内の応用では，アドレスは2バイトで指定可能であり，以上のような注意も必要ですが，アセンブリ言語でのプログラミングでは，68000の内部が"完全な32ビットマシン" であることから，アドレスレジスタのサイズを16ビットにしたところで処理スピードが高速になることもありません．このようなことから，68000のアドレスレジスタには，32ビットのアドレスをセットするものと解釈してもよいでしょうし，こうした方がスッキリすると思われます．

2. ロングワードサイズとしてのアドレスレジスタ

32ビットのすべてが操作対象となります．

3. インデックスレジスタとしてのアドレスレジスタ

扱えるオペランドサイズはワードとロングワードサイズであり，摘要範囲も各サイズに従います．そこで，ワードサイズであれば，符号拡張した32ビットの数値として扱われ，ロングワードサイズなら，32ビットのすべてが使用されます．

データレジスタ［D0～D7］図1.3

先のアドレスレジスタは，主にメモリを操作する時に必要なアドレスを出力するために使用されますが，本レジスタはメモリから取り出されたデータの加工や，メモリへ記憶されるべきデータの加工をするために使用され，一般に演算用レジスタとなります．

データレジスタはデータをMPU内に記憶しておくためにあり，D0～D7の8本が用意され，いずれも32ビット長です．

ハードウエア的にはデータバスとデータレジスタが接続されているものと解釈してよいのですが，データバスは16ビットでデータレジスタは32ビットになっているので，16ビットのバスで32ビットのデータをどう操作するかですが，これはMPU内部で適切な処理をしていますから，この点に関しては心配無用です．

8086のようにレジスタを上位と下位に分離し，それぞれ別々のレジスタとして指定し使用することはサポートされませんが，8本ものレジスタで不足することは極めて稀だし，レジスタの本数も8086よりは圧倒的に豊富です．仮にレジスタが不足するようなことがあっても，強力なアドレッシングモードがサポートされていることから，良好なプログラミング環境は依然保持されます．

図1.3　データレジスタの構成

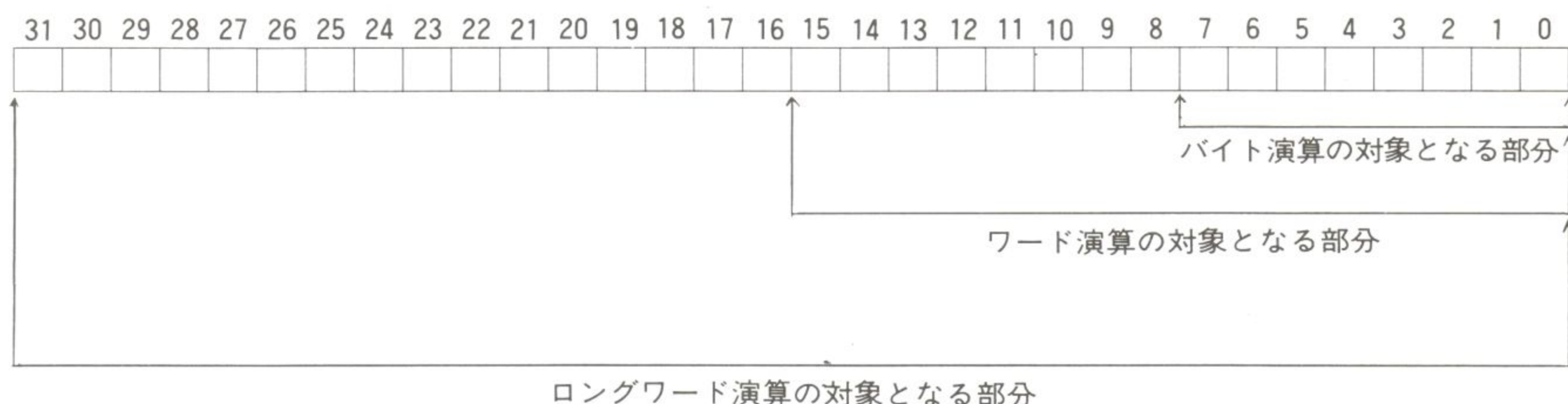
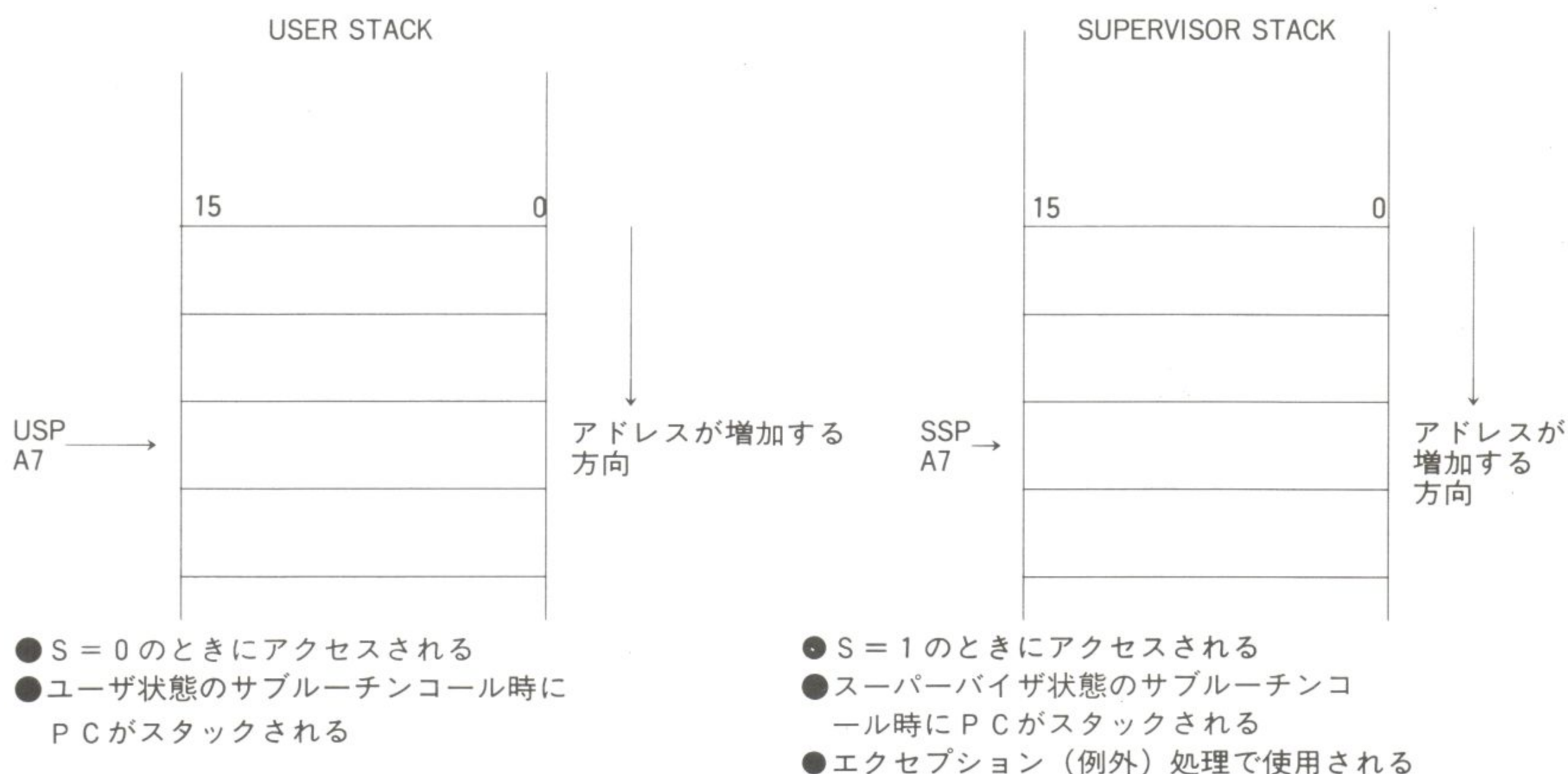



図1.4　システムスタック

【データレジスタとオペランド・サイズ】

データレジスタに対する操作は，すべてのサイズが許されます．

1.　ビット，バイト，ワード，ロングワードとしてのデータレジスタ

データレジスタでは，ビット、バイト，ワード，ロングワードなど，どのデータサイズでも演算を行うことが可能であり，バイト演算では最下位の８ビットが，ワード演算では下位16ビットが使用され，いずれの場合も残りの上位ビットはまったく無視され，無視されたこれらの各ビットの内容が変更されることはありません．したがって，データレジスタをソース・オペランドまたはディスティネーション・オペランドとして使用する場合，該当する下位部分のみが対象となり，残りの上位部分は無視され変化しません．

ロングワードでは32ビットのすべてが演算の対象となり，ビット０～31のすべてが使用されます．

2.　インデックスレジスタとしてのデータレジスタ

アドレスレジスタがインデックスレジスタに使えるのは当然ですが，データレジスタも

インデックスレジスタとして使用可能であり，より複雑なデータ構造を容易に操作することができます.

オペランドサイズはアドレスレジスタと同様であり，ワードサイズであれば符号拡張された32ビットの数値として使用され，ロングワードなら32ビット全体が使用されます.

システム・スタックポインタ［A7，SSP，USP］

アドレスレジスタの7番であるA7はシステム・スタックポインタに割り当てられます. A7もアドレスレジスタには相違なく，プログラミングレベルではA0～A6に対する命令とA7とは同レベルですが，その役割は明らかに異なっています.

① システム・スタックポインタは，プログラムが走行する際に必要な「特別なアドレス」，つまり，システムスタック領域をポイントするアドレスレジスタである.
② プロセッサスタート時に適切な値に初期化された後，プログラミングレベルで，他のアドレスレジスタのように頻繁に内容を書き換えることは要求されない.
③ 暗黙のうちに参照される典型的なレジスタであり，プログラマのために陰で働いている重要なアドレスレジスタである.

このように，A0～A6までのアドレスレジスタとは明らかにその役割が異なり，プログラマ自身がこれらのアドレスレジスタと明確に区別してプログラミングします. というよりは，通常はA0～A6を相手にプログラミングする，と表現すべきでしょう.

とりあえず，A0～A6までのアドレスレジスタとはその役割が異なること，さらにその役割は大変重要であることを知っておいてください.

なおアセンブラでは，A7（SSP，USP）を単に〝SP〟と記述できます.

【システムスタックとパラメータスタック】

システムスタック領域とパラメータスタック（ユーザスタック）領域とがどう異なるかですが，システムスタックはA7で管理される領域であり，ステータスレジスタのSビットによりSSPまたはUSPが使用されパラメータスタック領域はA7以外のアドレスレジスタであるA0～A6を使用してアクセスする領域を意味します.

システムスタックへはプログラムカウンタやステータスレジスタの内容や引数が，パラメータスタックへは引数（パラメータ）というように，操作される内容にも大きな違いがあります。

以上のような68000のスタックは，次のように分類することができます.

システムスタック	SSP／A7／で管理されるスタック領域（スーパバイザ状態）
	USP／A7／で管理されるスタック領域（ユーザ状態）
パラメータスタック	スーパバイザ／ユーザ状態に関係なくA0～A6で管理されるスタック領域

1. システムスタック（図1.4）

68000内部にはシステム・スタックポインタが2セット用意され，ステータスレジスタのSビット（ビット13）の状態に応じて，スーパーバイザ・スタックポインタ（SSP）また

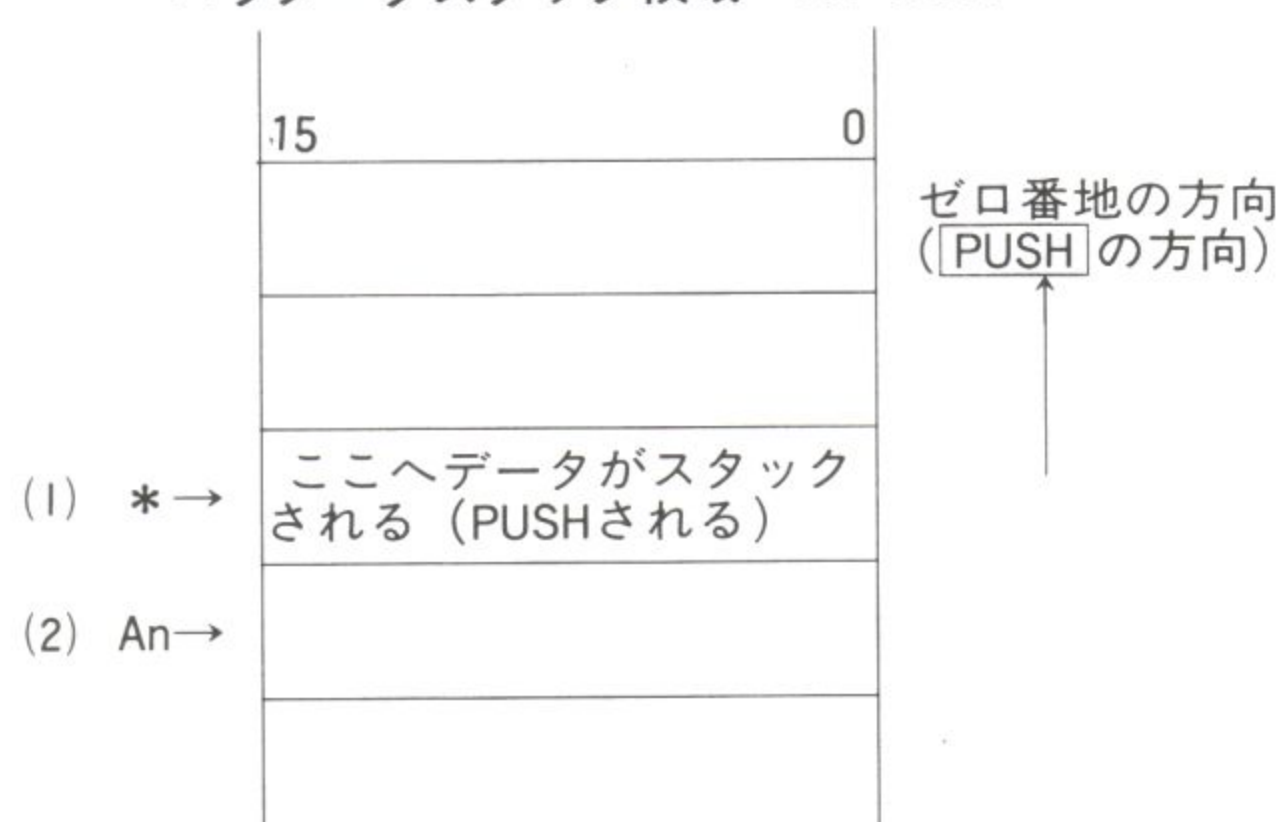
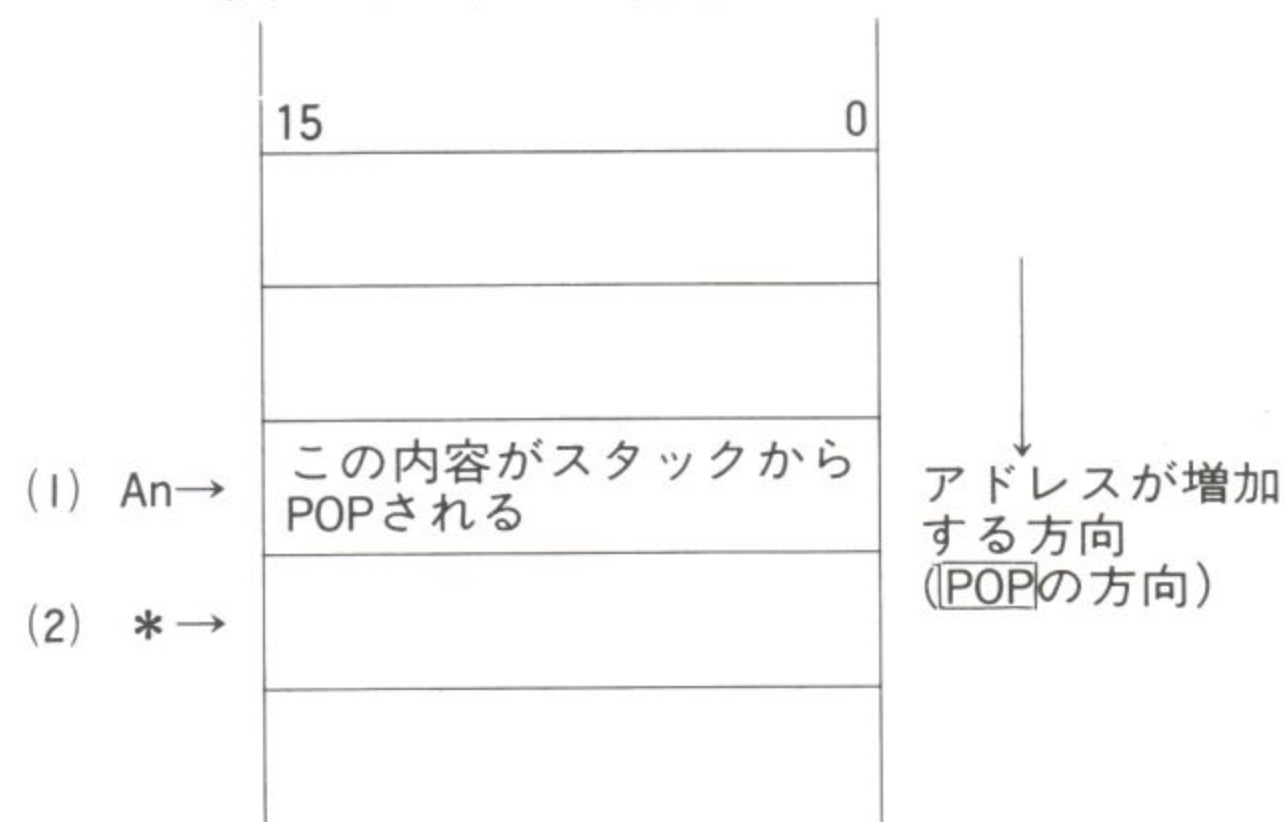
はユーザ・スタックポインタ（USP）のいずれかが使われます（先の通り，アドレスレジスタ番号としてはＡ７です）．

すなわち，ＳビットがON（１）ならば，SSPとしてのＡ７がアクティブスタックとして使用され，USPが参照されることはありません．同様にＳビットがOFF（０）ならば，USPとしてのＡ７がアクティブスタックとして使用され，SSPが参照されることはありません．

① ユーザスタック空間とスーパバイザスタック空間は独立したスタックポインタ（アドレスレジスタの７番であるＡ７）で管理される．
② システムスタック領域は，メモリアドレスの大きい番地からゼロ番地へ向かって割り当てられる（スタックはゼロ番地方向へ向かって成長する）．
③ プリデクリメント・アドレッシングモード　〝－（SP）〟　によって，アクティブなシステムスタック上に新しいデータを追加する．これを一般にプッシュ（PUSH）という．
④ ポストインクリメイト・アドレッシングモード　〝(SP)＋〟によって，アクティブなシステムスタック上からデータを取り出す．これを一般にポップ(POP)という．
⑤ スタック領域は常にワードの境界単位でアクセスされるので，バイトデータをシステムスタックへプッシュしたりポップしたりする場合，ワードの上位半分で行われ，下位半分は変化しない．
⑥ サブルーチンコールの際には，プログラムカウンタはアクティブスタック領域へ退避され(プッシュ)，リターン時にアクティブスタック領域から復帰（ポップ）される．
⑦ トラップ処理中または割り込み処理中には，プログラムカウンタとステータスレジスタが，スーパーバイザスタック領域へ退避（プッシュ）される．

2．パラメータスタック（ユーザスタック）図1.5

パラメータスタックとは，プログラム中で意識的に操作するスタック領域を意味します．

図1.5　PUSHとPOP

●サイズはワードを例にしている
(1)Anがポイントしているアドレスであり、命令実行前はこのようになっている
(2)命令実行後Anはデクリメントされ図＊のようになる．たとえばMOVE．W　D0，－(A0)を実行したとき．

●サイズはワードを例にしている
(1)Anでポイントしているアドレスからデータを取り出す（POPする）
(2)POP後、An をインクリメントして図＊のようになる．たとえばMOVE．W　(A0)＋，D0を実行したとき．

システムスタック領域は，従来のプロセッサでも当然サポートされていたわけですが，68000ではパラメータスタックさえも単なるデータ領域にすぎず，非常に柔軟なスタック操作を支援しています．

スタック操作であるPUSHやPOPには次のようなアドレッシングを適用し，〝n〞には0～7までのアドレスレジスタ番号を指定できます（A7はシステムスタックをポイントする）．

－（An）　：PUSH操作（プリデクリメント・アドレッシングモード）
（An）＋　：POP操作　（ポストインクリメント・アドレッシングモード）

スタック操作でのPUSHとPOPは，一般的には対（ペア）で使用され，PUSH操作だけとかPOP操作だけというのではなく，まずPUSHがあって次にPOPが意味を持つことを理解しておいてください（試しに，PUSH操作，次にPOP操作をしてみてください．ちゃんと元通りにスタックが復元されるのを確認できるでしょう）．

① －(An)を使用する場合は，指定したAnの内容をデクリメントしてアドレスを操作し，その内容をスタックへのポインタとする（〝－〞記号が先に付加されていることに注目）．
② (An)＋を使用する場合は，指定したAnの内容をスタックへのポインタとし，データを取り出した後でAnをインクリメントする（〝＋〞記号が後に付加されていることに注目）．
③ 意識的なスタック操作であってもA7（SP）がポインタなら，スタックは常にワード（偶数）の境界でアクセスされるので，指定したアドレスレジスタの内容は必ず偶数に保持されます．そこで，バイトデータに対するスタック操作であっても，ワードとして扱われます．

プログラムカウンタ［PC］

プログラムカウンタは，次に実行すべき命令が格納されているメモリアドレスを保持する特別なアドレスレジスタであり，プロセッサはPCの値をたよりに一連のプログラムを実行していきます．

PCは32ビットで構成されますが，サポートされるのはビット0～23までの24ビットであり，最下位のアドレスバスA0は上位バイトと下位バイトとの区別にMPU内部で使用され，外部にはアドレスバスのA1～A23が出力されています．この点ではA0～A7までのアドレスレジスタと同様です．

68000には命令長がワード単位であるという明確な設計思想があり，8086がゴチャゴチャした構成であるのに比較すると，アーキテクチャに無理がありません．このような理由から，68000の命令コードは必ず偶数番地から配置されますので，PCの内容も偶数ということになり，奇数であるとアドレスエラー例外が発生します．

このアドレスエラーもシステムの信頼性を支えており，どこであろうと，PCでポイントされる場所に命令が格納されているものとして処理する従来のマイクロプロセッサに比較すれば，明らかに高級なマイクロプロセッサといえます．

ステータスレジスタ［SR］

ステータスレジスタは16ビットで構成されますが，ビットマップになっており，

プロセッサ自身に関する特別な制御に使用され，上位バイトをシステムバイト，下位バイトをユーザバイトまたはCCR（コンディション・コード・レジスタ）と呼びます．

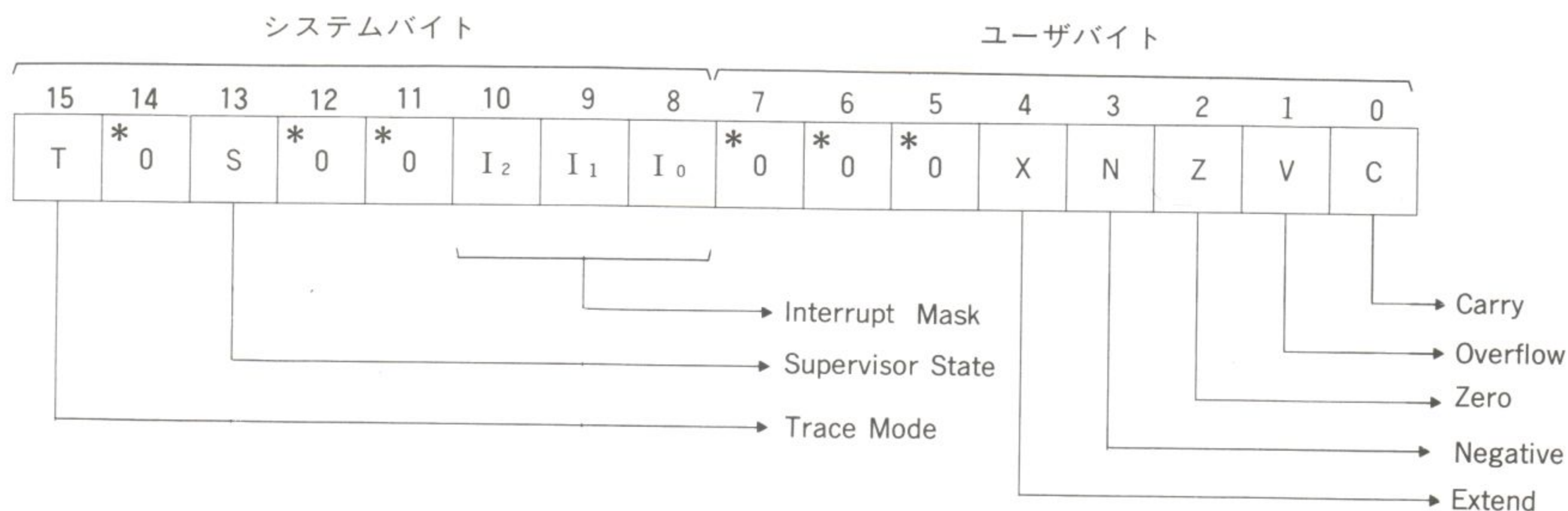
図1.6　SR（ステータスレジスタ）の構成

＊ゼロ（0）が代入されているビットは、68000では未使用である
　その他シンボルが記入されているビットは"意味"のあるビットである

表1.1　システムバイト（SRのビット15～8）

シンボル	SRビット	説　　明
T	15	"1"にセットされていると，命令実行ごとにスーパーバイザ状態へ移行し，トレース処理を行う．トレース・モードはプログラムのデバックをサポートする機能．
S	13	スーパーバイザ状態なのか（S=1），ユーザ状態（S=0）なのかを保持している．
I_2, I_1, I_0	10,9,8	この3ビットで割り込みのマスクをするが，7レベルのうちのどのレベルに設定されているかを保持している．

表1.2　ユーザバイト（コンディション・コードレジスタ：CCR）

シンボル	SRビット	説　　明
X	4	エクステンド（拡張）を意味する"X"であり，多倍長演算（高精度演算）に使用されるフラグ．キャリと同様な意味であるが，本フラグの方が命令の適用範囲が狭い．
N	3	ネガティブ（負）を意味する"N"であり，演算結果が負になるとセット（1）される．
Z	2	ゼロ・フラグを意味する"Z"であり，演算結果がゼロになるとセット（1）される．
V	1	オーバーフロー・フラグを意味する"V"であり，オーバーフローが発生するとセット（1）される．
C	0	キャリ・フラグを意味する"C"であり，演算結果で桁上がり（キャリ），あるいは減算でボロー（桁下がり）が発生するとセット（1）される．

以上で68000内部のレジスタの説明を終わりますが，68000自身にとって必要だから"ある"のであって，もし，これらのレジスタがなければどうなるかという単純な疑問を持てば，各レジスタの役割を理解するのも容易であろうと思われます．

4 メモリ内のデータ構成

[1] 予備知識としての"サイズ"

端的に，プログラミングとはデータの操作（加工）を意味し，このデータとはメモリ内に記憶されています．二次的な記憶媒体（ディスクなど）上のデータであっても，結局メモリに読み込まれて操作されるので，我々が操作しようとしているデータが，「どのような約束でメモリ内に記憶されるのか」を把握しておかねばなりません．

ご承知のように，数値であれ文字であれコンピュータ内部では〝1〞と〝0〞で表現され，8ビットなら〝1〞と〝0〞の組み合わせが2^8通りとなり，符号付バイナリ値なら－128～＋127の256通り，符号なしバイナリ値なら0～255までの256通りの表現しかできません．

これらのデータサイズは，プログラミング言語がどのようなものであれ，必ず〝型〞の宣言としておなじみのものであり，高級言語でも〝型の宣言〞は強要されます．
コンピューターがデータを扱うためには，〝サイズ〞という属性が必要であり，これが我々にとっては〝制約〞となっているわけです．

[2] 68000のサポートするメモリ内のデータ構成

68000がデータをメモリへ書き込んだり，読み出したりするときのアクセスの単位は，バイト，ワード，ロングワードの3つがサポートされます．

データサイズの諸元を**表1.3**に示します．

表1.3　68000のデータサイズの諸元

データサイズ	バイト数	ビット長
バイト	1バイト	8ビット
ワード	2バイト	16ビット
ロングワード	4バイト	32ビット

一般には，これら3つのデータサイズが1個のデータとして記憶されることはなく，アルファベットによる名簿管理なら1バイトのデータが集まって一人分の文字列を構成し，漢字表現の文章なら1文字は1ワードで表現され，これらが集まって文を構成する，というようにメモリ内へ記憶されます．

この結果，ある連続したメモリ空間を占有することになり，メモリアドレスの上位と下位という関係も把握しておかねばなりません．このようなことを理解しておけば，データのアクセスは必要に応じて（プログラミングを意味する）混同できるので，バイトで記憶したデータをワードで取り出すことも，ロングワードで取り出すことも可能であり，逆に，ワードやロングワードで記憶した内容をバイト単位で取り出して操作することもできます．

以下に各データ（バイト，ワード，ロングワード，BCD，アドレス）がどのようにしてメモリ内へ格納されるのかを説明しますが，単に「このようになっているのか」と納得するのではなく，先の例のように，ある種のモデルをイメージしてみることは大切です．

バイトデータがメモリへ記憶される様子 図1.7

ビット番号はビット7～0までの8ビットであり，ビット7が最上位ビット（MSB）で，ビット0が最下位ビット（LSB）となります．

68000は16ビットのデータバスを備えており，偶数番地のメモリアドレスにはD15～D8のデータバスが，奇数番地にはD7～D0のデータバスがインターフェースされ，偶数番

図1.7 メモリへ格納されるバイトデータ

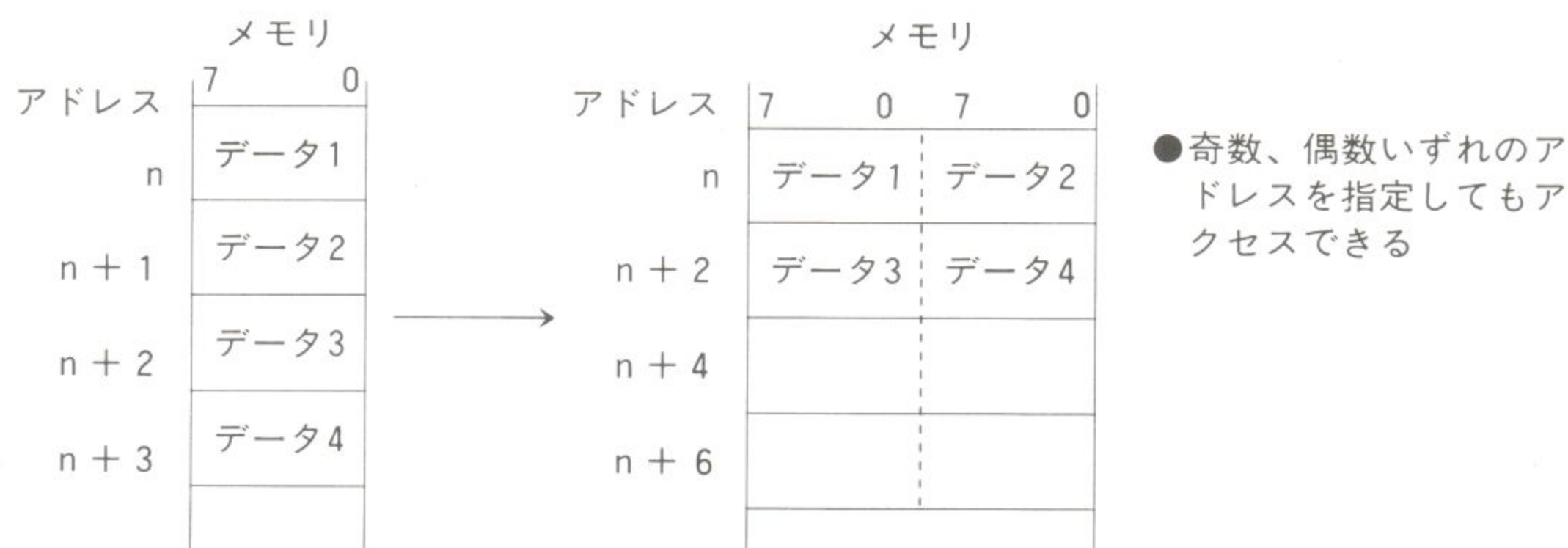



図1.8 メモリへ格納されるワードデータ

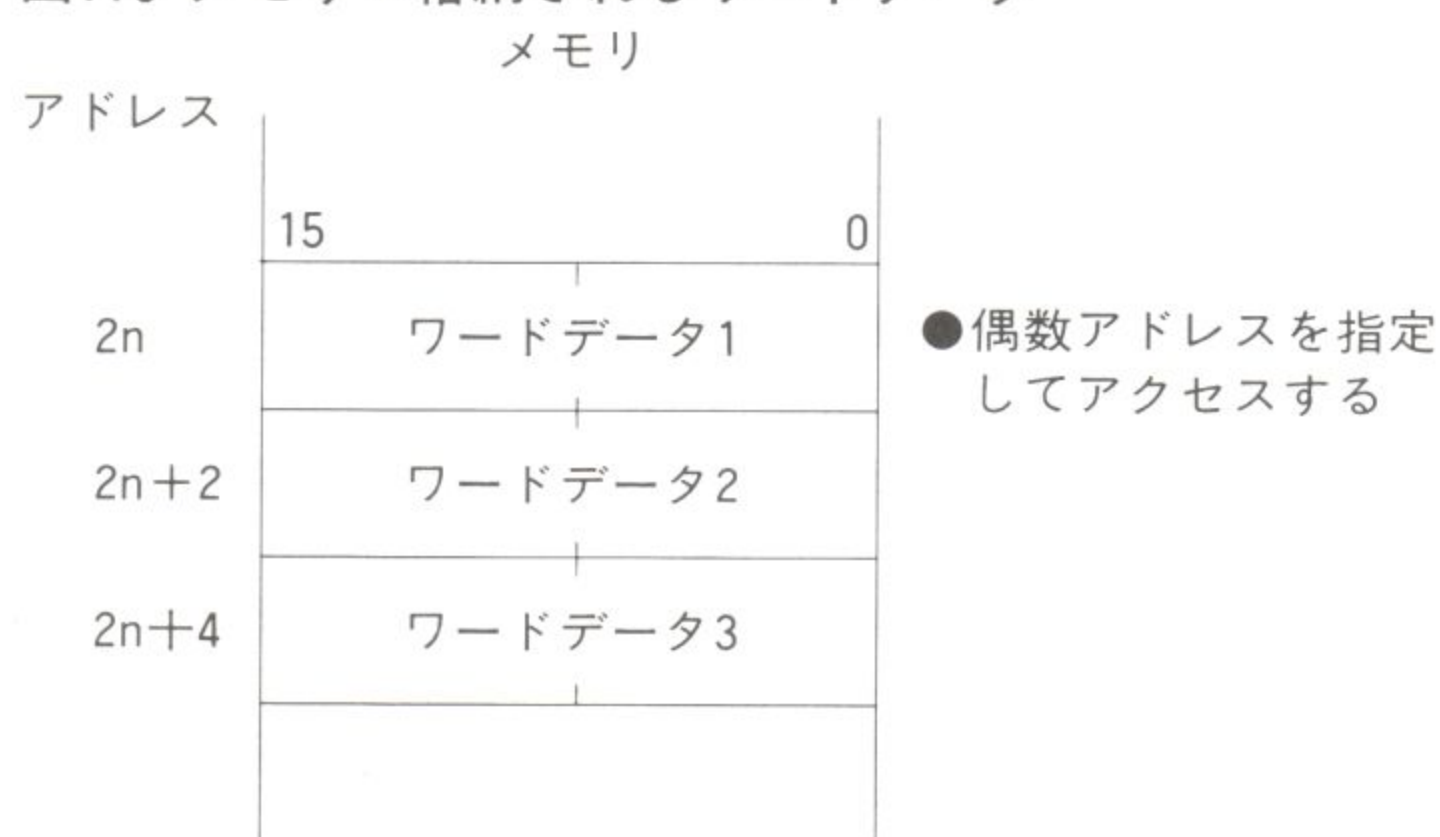



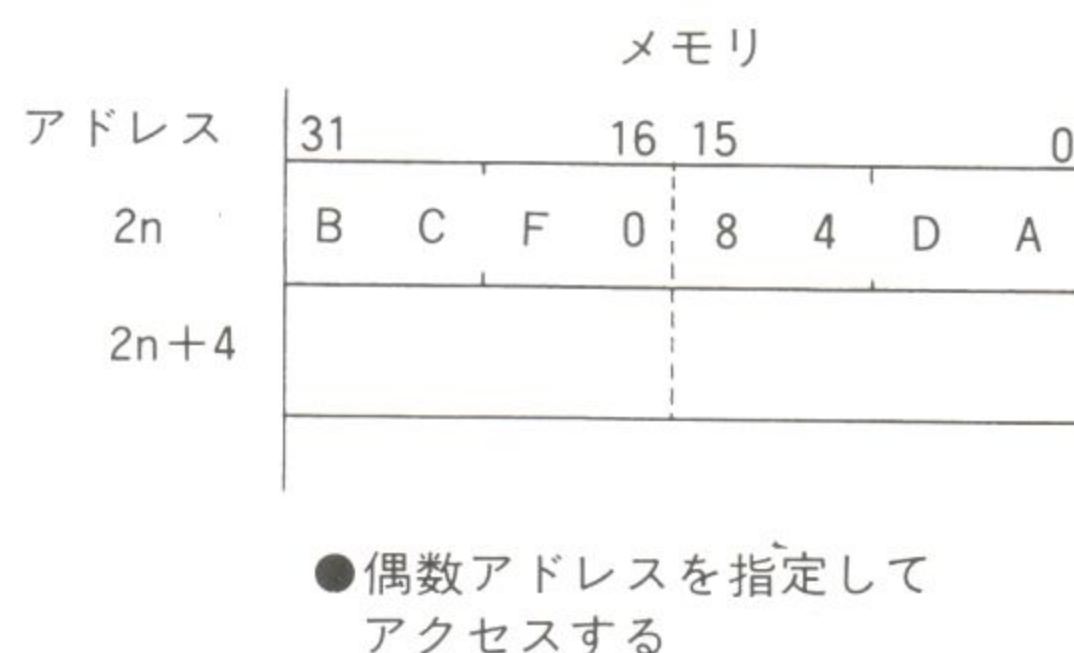
図1.9 メモリへ格納される ロングワードデータ

地が上位バイト，奇数番地が下位バイトに割り当てられています．

図1.7ではバイトデータを扱うことを明示する意味で，ビット番号は7～0になっていますが，たとえば，偶数番地のバイトデータをデータレジスタへ読み込む場合，データバスの上位であるD15～D8を使用してアクセスし，その内容がデータレジスタの所定位置であるビット7～0へ格納されます．同様に奇数番地のバイトデータをデータレジスタへ読み込む場合，データバスのD7～D0を使用し，データレジスタのビット7～0へ格納します．データが逆方向へ転送される場合もまったく同様です．

このように，バイトデータは偶数／奇数アドレスを問わず，アドレスレジスタで指定された番地を個別にアクセスできます．

ワードデータがメモリへ記憶される様子 図1.8

ワードデータは2バイトですから，ビットマップではビット15～0であり，ビット15がMSB，ビット0がLSBであり，ビット番号はデータバスのビット番号に対応します．

バイトマップで見た場合，上位バイトは偶数番地，下位バイトは奇数番地に配置されますが，アクセス時には必ず偶数番地を指定する必要があり，指定した偶数番地からアドレスの増加する方向へ，連続した2バイトが処理の対象となります．奇数アドレスから連続した2バイトをアクセスすることは禁止されています．

ロングワードデータがメモリへ記憶される様子 図1.9

ロングワードデータはバイトマップでは4バイト，ワードマップでは2ワードから構成され，ビットマップではビット31～0であり，ビット31がMSB，ビット0がLSBです．

表1.4　バイトマップの場合

アドレス	内　　容	備　　考
2ｎ	$BC	最上位バイト
2ｎ+1	$F0	
2ｎ+2	$84	
2ｎ+3	$DA	最下位バイト

表1.5　ワードマップの場合

アドレス	内　　容	備　　考
2ｎ	$BCF0	上位ワード
2ｎ+2	$84DA	下位ワード

アクセスはワードの境界で行われ，偶数アドレスを指定すると，そこからアドレスの増加する方向へ連続している4バイトが処理の対象になります．

今，16進数の　$BCF084DA　がアドレス2ｎ番地から格納される場合，**表1.4**，**表1.5**のような各データが格納されることになります．

そこでロングワードが記憶されているデータを，ワード単位で操作する必要がある場合には，上位ワードならアドレス2ｎを指定し，下位ワードならアドレス2ｎ+2を指定してアクセスできますが，奇数アドレスを指定してはなりません．もちろん，バイトアクセスであれば，4バイトのうちの任意バイトを操作することができます．

参考までに8086フォーマットで　$BCF084DA　がメモリへ格納される様子を**図1.10**に示します．

次のように分解します．

① 上位ワードは　$BCF0　である．
② 下位ワードは　$84DA　である
③ アドレス2ｎ，2ｎ+1には下位ワードの上位／下位が交換されて格納されるので，それぞれ，$DA，$84，が順に格納される．
④ アドレス2ｎ+2，2ｎ+3には上位ワードの上位／下位が交換されて格納されるので，それぞれ，$F0，$BC，が順に格納される．

$F0を見て即座に上位ワードの下位バイトであると判断しにくいし，下位ワードが$84DAであることを判断するのも重荷です．1ロングワードだけなら逆に配置されているだけですから，後ろからたどれますが，ワードデータとロングワードデータが2～3個も連続すると，ワードの境界があやふやになり，デバッグ時のメモリダンプや特定の1バイトだけを書き換えたい時にミスを犯しやすくなります．

これは，我々が「上から下」，「左から右」へ向かって数値や文章を記述する文化（習慣）と異なっているからです．68000のデータ形式は8086とは異なり，極めて自然なフォーマットで表現されるわけです．

ワードまたはロングワードデータは，必ず偶数アドレスから配置されることを述べましたが，奇数アドレスから配置できないことがディメリットになることはなく，かえって混乱することがありません．一方8086では，偶数／奇数という制約はないのですが，それだけに混同するチャンスも多くなるかもしれませんから，データサイズの混乱抑制のためにも，〝あるスタイル〟を独自に確立すべきでしょう．

BCD (Binary Coded Decimal) データ　図1.11

BCDは2進化10進を意味し，16進の$0～$Fのうちの$A～$Fの部分を使用せず，$0～$9までを使用して10進表現をするもので，10進演算に使用されます．

コードは4ビットで表現されますから，1バイト中に2桁を格納することができます．

➡32ページへ続く

図1.10 ロングワード（ダブルワード）$BCF084DAと8086フォーマット

図1.11 メモリへ格納されるBCDデータ

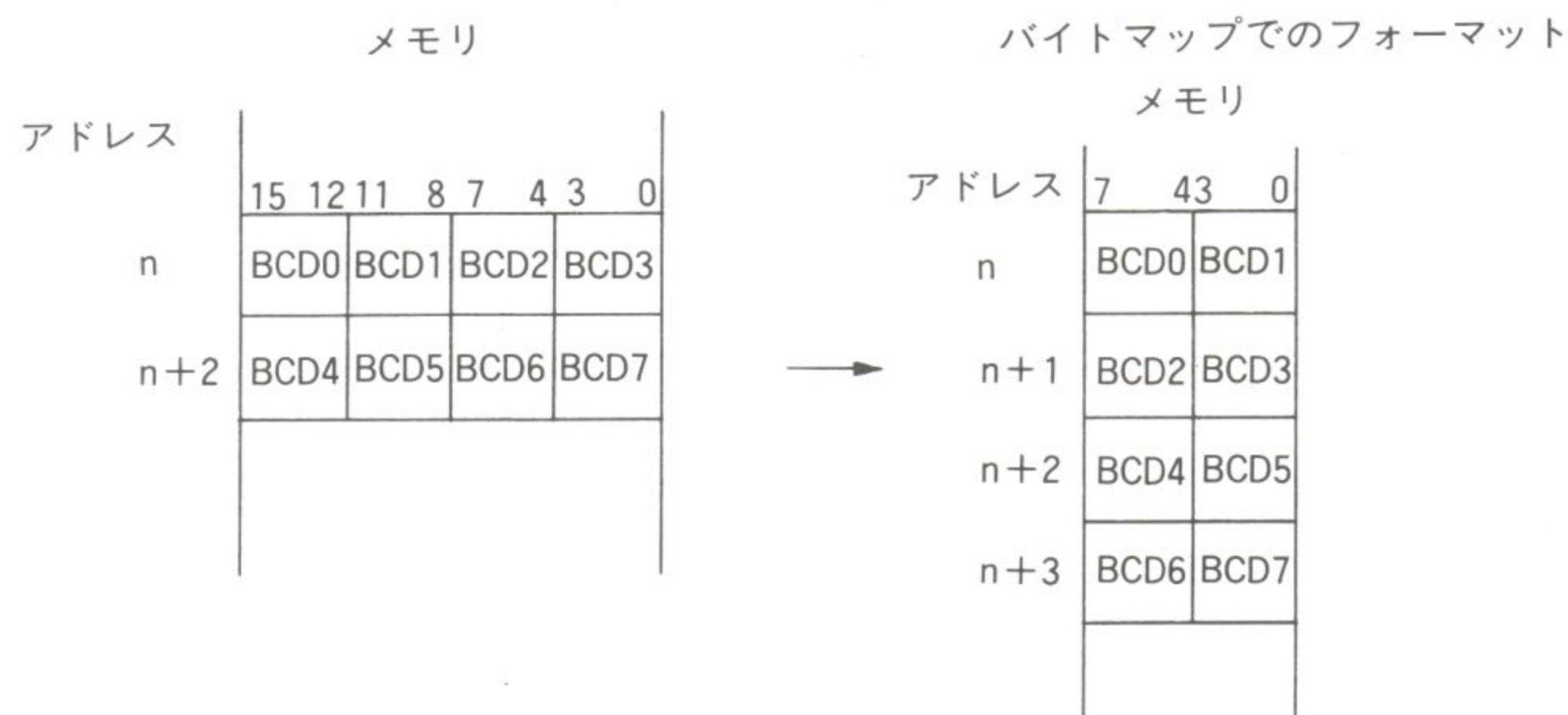



表1.6 図1.11のバイトマップ

アドレス	内容	備考
n	BCD 0，BCD 1	BCD 0は最上位（1バイトで2桁を表現する）
n ＋ 1	BCD 2，BCD 3	
n ＋ 2	BCD 4，BCD 5	
n ＋ 3	BCD 6，BCD 7	BCD 7は最下位

図1.12 メモリへ格納されるアドレスデータ

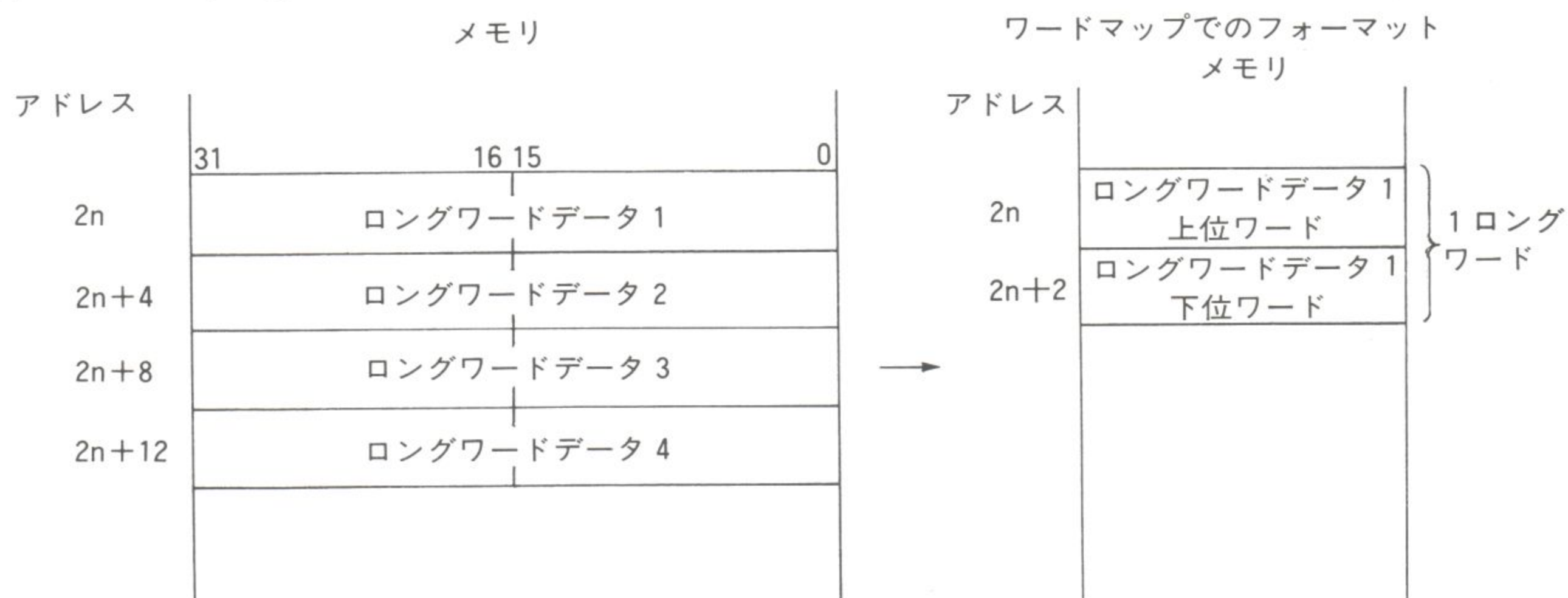



図1.11では２ワード中（連続した４番地）にBCDデータの８桁がどのように格納されるかを示します．BCD 0 が最上位の桁でありBCD 7 が最下位の桁ですが，バイトマップでは**表1.6**のようにマップされます．

アドレスデータ 図1.12

アドレスデータは常にロングワード（32ビット／４バイト）としてメモリへ格納され，指定された偶数番地からの４バイトがアクセスの単位となります．メモリ上のフォーマットは先のロングワードデータと同一ですから，必ず偶数アドレスを指定しなければなりません．

アドレスデータ　＄88DFB０Ｂ１　は，**表1.7**，**表1.8**のようなフォーマットになります．

表1.7　バイトマップの場合

アドレス	内　容	備　考
２ｎ	＄88	最上位バイト
２ｎ＋１	＄DF	
２ｎ＋２	＄B0	
２ｎ＋３	＄B1	最下位バイト

表1.8　ワードマップの場合

アドレス	内　容	備　考
２ｎ	＄88DF	上位ワード
２ｎ＋２	＄B0B1	下位ワード

データフォーマットに関しては，以下のような事柄に留意されるとよいでしょう．

▶ 68000のサポートするデータが，メモリ上でどのようにフォーマットされるかということは，アセンブラでプログラミングする際には必修事項であること．

さらに，すべてのデータサイズにおいて，「上から下」，「左から右」というような極めて自然なフォーマットであり，バイト～ワード～ロングワード間の変換もまた極めて自然に変換することが可能であること．

なお本書ではアドレスは「上から下」に向かって増加するように記述するが，この表記に関しては統一されてないので，「下から上」に向かってアドレスが増加するように表記されることも，しばしば見うけられる．

▶ プログラミング過程では，扱う対象をどのようにしてメモリへ記憶するのか，あるいは，どのようにして操作するのかということを明確にするため，必ず図示してみる．この点に関しては，必要に応じてバイト／ワード／ロングワードのフォーマットで表現すべきだが，68000が16ビットのプロセッサであることから，通常はワードマップにし，その中でバイトやロングワードをイメージするとよい．

5 命令に関する基本事項

このセクションでは，命令そのものを理解する上で必要な基本的事柄について説明しますが，命令の動作がよく理解できない場合には，何時でも振り返る必要があります．

最初に68000が命令を実行するときに参照するメモリ区分について，次に機械語の構成，アドレッシングに関する説明がなされています．このうちで特に重要なのはアドレッシングモードであり，アセンブラでプログラミングする上で，どうしても理解しておかねばなりません．

[1] プログラム領域とデータ領域の参照

どのようなプロセッサでもメモリ上のデータは2つの属性を持っています．

すなわち，プログラムを意味するデータとそれ以外のデータですが，プログラム領域は本質的にリードオンリ（読み出し専用）です．非常に特殊な用途を除き，プログラム自身を実行中に書き換えることは，プログラムとしての機能を失い，結果的に暴走することになるのは，いうまでもないことです（ただし68000には例外処理という強力な機能がある）．

68000が両者をどのようにして区別しているかというと，MPU内のPCで指定されるメモリ領域にはプログラムが格納され，我々が操作するメモリ領域はこの領域とは別のデータ領域であり，アドレスレジスタへセットするメモリ空間がデータ領域である，と思ってもよいでしょう．具体的な輪郭はアセンブラの経験を積むに従って明確になってきますから，この段階で深刻になることはありません．

以下のように要約できます．

① メモリへの参照はプログラム参照とデータ参照に分類され、前者はプログラムが格納されているメモリ領域への参照であり，後者はデータの格納される領域への参照である．

② プログラムカウンタ相対形式のアドレッシングモードを除き，オペランドのリードはデータ領域から行われ，すべてのオペランドの書き込みはデータ領域に対して行われる．

[2] 機械語の構造

この項目はアセンブラや逆アセンブラなどのツール開発に必要な情報であり，これを知らなくても，特にプログラム開発に影響するわけではありません．しかし，購入したツールが予定の行動をしていないような場合には，出力されたオブジェクトをチェックしなければなりません．このような状況下では，マシン語のフォーマットを理解していなければならないので，知っていて損になるようなものでもありません．

68000には3つの機械語フォーマットがあり，我々が記述したアセンブラのニーモニックは，これらのいずれかの形式をとり，1～5ワードまでの機械語となって（正しくは，1～5ワードにアセンブルされて）メモリへ格納され，実行されることになります．

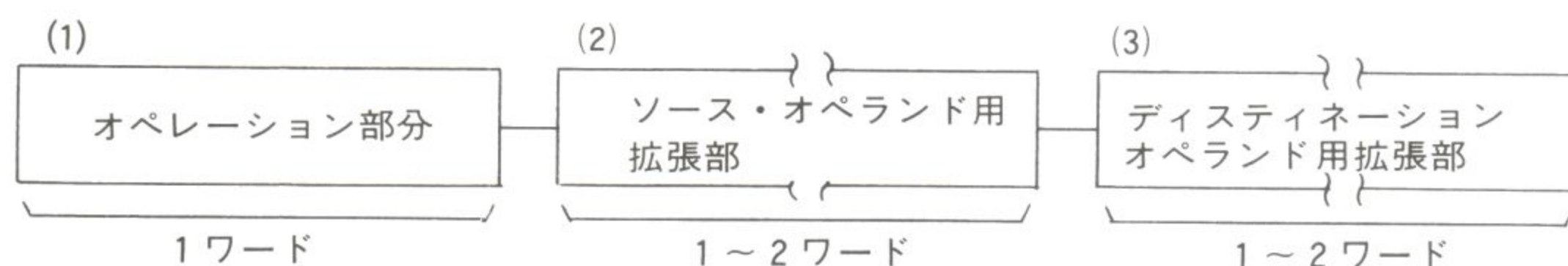
各拡張部はオペランドの指定方法により，必要に応じて1ワードか2ワードで構成されますが，これらの拡張部をまったく必要としない命令も存在します．したがって，1つの命令から得られる機械語のサイズは，1ワード（2バイト）から最大長5ワード（10バイト）までであり，これ以外の機械語にアセンブルされることはありません．

各フォーマットの構成要素番号で示されるサイズはワード（2バイト）であり，番号の小さい方が上位ワードであり，上位ワードから順にメモリ上へ格納されます．

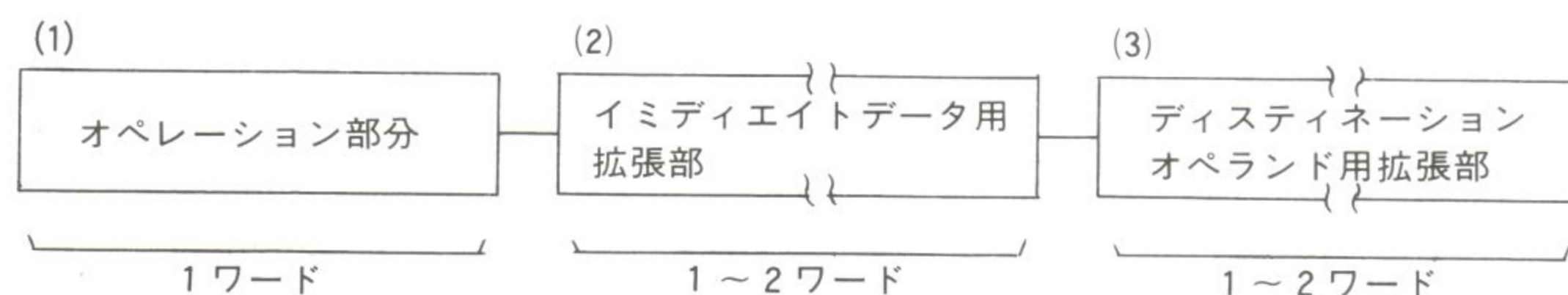
機械語フォーマットはアドレッシングモードとも密接な関係にあるわけですが，最終的には個別命令のセクションで詳細に展開されるので，ここでは，以下の3つのフォーマットに分類されることを把握すれば十分でしょう．

機械語フォーマット●1

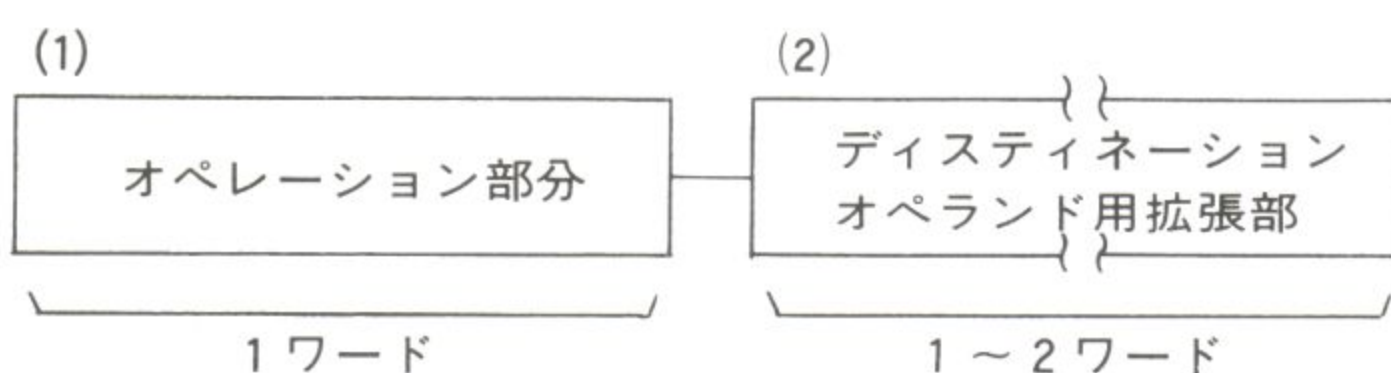
① オペレーション部を意味するフィールドで1ワードで構成される．
② ソース・オペランド用拡張部で1～2ワードで構成される．
③ ディスティネーション・オペランド用拡張部で1～2ワードで構成される．

図1.13 機械語フォーマット1

機械語フォーマット●2

① オペレーション部分を意味するフィールドで1ワードで構成される．
② イミディエート・データ（即値）用拡張部で1～2ワードで構成される．
③ ディスティネーション・オペランド用拡張部で1～2ワードで構成される．

図1.14 機械語フォーマット2

機械語フォーマット●3

① オペレーション部分を意味するフィールドで1ワードで構成される．
② ディスティネーション・オペランド用拡張部で1～2ワードで構成される．

図1.15 機械語フォーマット3

6 アドレッシング

[1] アドレッシングについて

アドレッシングは直接プログラミング環境そのものを支配するもので，これを理解してしまえば，アセンブラでプログラミングする際に必要な知識のほとんどを得たことになります．

プロセッサのサポートする基本命令は，内部のレジスタや扱うデータの構成が異なる程度で，どのようなプロセッサでも同じような命令セットになっているからです．

これから具体的なアドレッシングモードの説明に入りますが，非常に強力なアドレッシングがサポートされ，Z－80や8086での苦労は何であったのかと思いたくなります．つまり，アーキテクチャの進歩はプログラミングの容易さ，一方ではバグの発生しにくいプログラムの開発環境，高速な処理，などと密接な関係があり，マイクロプロセッサのパフォーマンスそのものを決定する要因の一つであるといえます．

この点で，「16ビットだから大変だ！」という心配は68000のプログラミングに関しては適当ではありません．アーキテクチャがよければ，プログラミング環境はそれだけ整備されたものになり，制約も軽減されるのですから………．筆者などは，アドレッシングモードの充実ぶりから，これなら「相当大きなプログラムまでアセンブラで記述できるナ」と直感しました．

アドレッシングとは〝アドレスの指定方法〟に他ならないわけですが，一般にはアドレスの指定以外にもレジスタの指定方法など，データへのアクセス方法全般を指します．

つまり，データがメモリにあってもMPU内のレジスタにあっても，そこへのアクセス方法を単にアドレッシングといい，レジスタに関する指定方法をレジスタ・アドレッシング，メモリに関する指定方法をメモリ・アドレッシングといいます．

最終的に決定されるメモリアドレスは実効アドレス（EA:Effective Address）と呼ばれますが，実効アドレスのバリエーションはアドレッシングでもあり，同じような扱いをうけます．

[2] 68000のサポートする実行アドレスモードの分類

実効アドレスモードは使われ方によって次のような4つのモードに分類されます．本書では，いきなり命令を使うことができるように，実用的な整理を各命令に対して行っているので，単にこのように分類できることを把握しておけばよいでしょう．

① **データ**：データとして，ビット，バイト，ワード，ロングワード，のすべてのサイズで扱うことが可能な形式．

② **メモリ**：データがMPUレジスタではなく，メモリ上にある形式．

③ **可変**：結果の記憶や値の変更など，操作の対象がMPUレジスタであろうとメモリ上にあろうと，書き換え操作可能な形式．

④ **制御**：メモリモードのうち，オペレーションサイズに関係しない形式．

これらのモードと実際のアドレッシングモードとの対応表を**表1.9**に示しますが，後に「データ可変」などと説明されれば，データモードと可変モード共通に含まれるアドレッシングモードを意味します．

表1.9　アドレッシングモードの分類

アドレッシングモード	モード			
	データ	メモリ	可変	制御
データレジスタ直接	○		○	
アドレスレジスタ直接			○	
アドレスレジスタ間接	○	○	○	○
ポストインクリメント アドレスレジスタ間接	○	○	○	
プリデクリメント アドレスレジスタ間接	○	○	○	
ディスプレースメント付き アドレスレジスタ間接	○	○	○	○
インデックス付 アドレスレジスタ間接	○	○	○	○
短または長絶対アドレス	○	○	○	○
ディスプレーメント付きプログラム カウンタ相対	○	○		○
インデックス付 プログラムカウンタ相対	○	○		○
イミディエイトデータ	○	○		
CCR／SR				

[3] アドレッシングモードの詳細

これから説明されるアドレッシングモードが，具体的にプログラミングをする上でどのような意味を持っているかを理解することは，アセンブラの初心者には少々大変かもしれません．経験者なら即座にこの部分を理解し，68000というマイクロプロセッサが提供するプログラミング環境の全体像を，容易にとらえることができるでしょう．それは，経験者はデータを扱う上で必要とされる〝あるモデル〟をイメージできるからです．

実例とともにアドレッシングモードの詳細を述べますが，コーディング例は必ずしも適切なものではなく，単にアドレッシングの説明に使用されるだけです．

例は「日立マイクロコンピュータシステム」に準拠しています．これは68000を設計する場合に，同様な例であれば混乱が少ないであろうと判断したからですが，プログラム・カウンタ・リラティブの表記などに，モトローラ社の文献の表記を採用しています．

なお，表記方法に関しては大差ないと思われますが，ツールの開発元から提供されるマニュアルで確認してください．

アドレッシングモードのポイント

すべてのア.ドレッシングモードに精通していなければプログラミングできないわけでは

なく，メモリ・アドレッシングに関する全体のおよそ7割程度は，次の3つのモードが多用される傾向にあります．

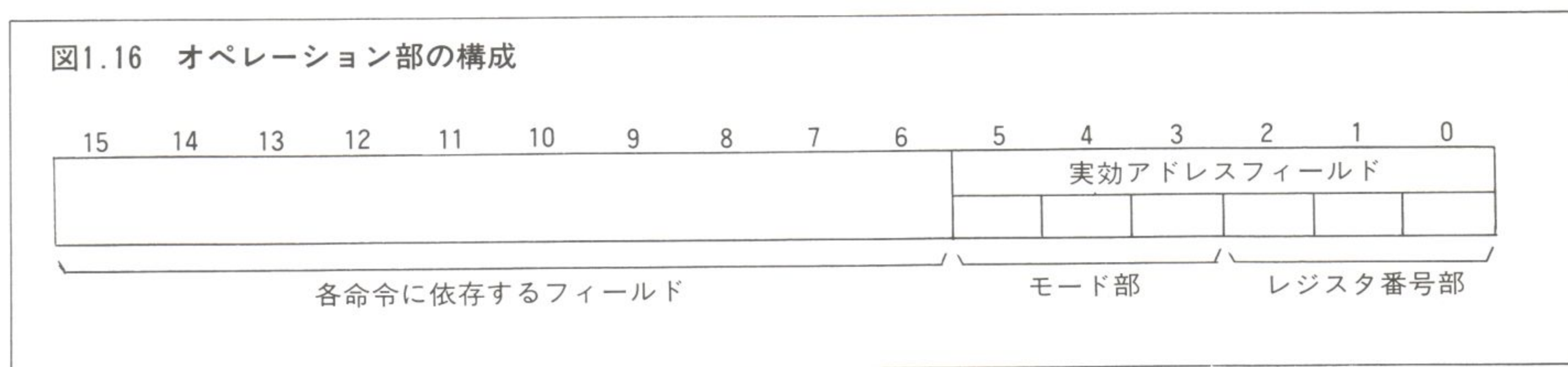
① (An) ：アドレスレジスタ間接形式
② (An)＋ ：ポストインクリメント・アドレスレジスタ間接形式
③ －(An) ：プリデクリメント・アドレスレジスタ間接形式

ですから，これらの使い方を理解するのが先決であり（といっても別に障害はないと思いますが………），その後，より複雑なアプリケーションに応じて，インデックス用法などをマスタするのがよいでしょう．それから，メモリ上の内容を操作する場合には，必ずメモリのイメージを図示することも大切です．

実効アドレスフィールド部を持つオペレーション部の構成

アドレッシングモードの説明をする前に，実効アドレスフィールドのフォーマットについて整理しておきます（**図1.16**）

図1.16 オペレーション部の構成

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
各命令に依存するフィールド										実効アドレスフィールド					
										モード部			レジスタ番号部		

① オペレーション部のビット15～6は各命令に依存するフィールドである．
② 実効アドレスフィールド部はビット5～0にマップされ，実効アドレスをオペランドに指定しない命令では，実効アドレスフィールド部は存在しない．
③ 実効アドレスフィールド部は，モード部とレジスタ番号部から構成され，実効アドレスフィールド部のビット5～3はアドレッシングモードの指定に，ビット2～0はレジスタ番号の指定にマップされる．

7 アドレッシング・モード

1 データレジスタ直接形式
(Data register direct)

アセンブラ書式: **Dn** [n=0~7]

解説 操作の対象は指定したデータレジスタであり，命令はそのレジスタの内容に対して直接実行されます．

MOVE.W D0, $1F00

ソース側はデータレジスタ直接，ディスティネーション側は絶対ショート形式によるもので，D0の下位ワードがアドレス$1F00から連続した2番地へ転送されるが，D0の上位ワードは無視される．

図1.17 データレジスタ直接形式の例

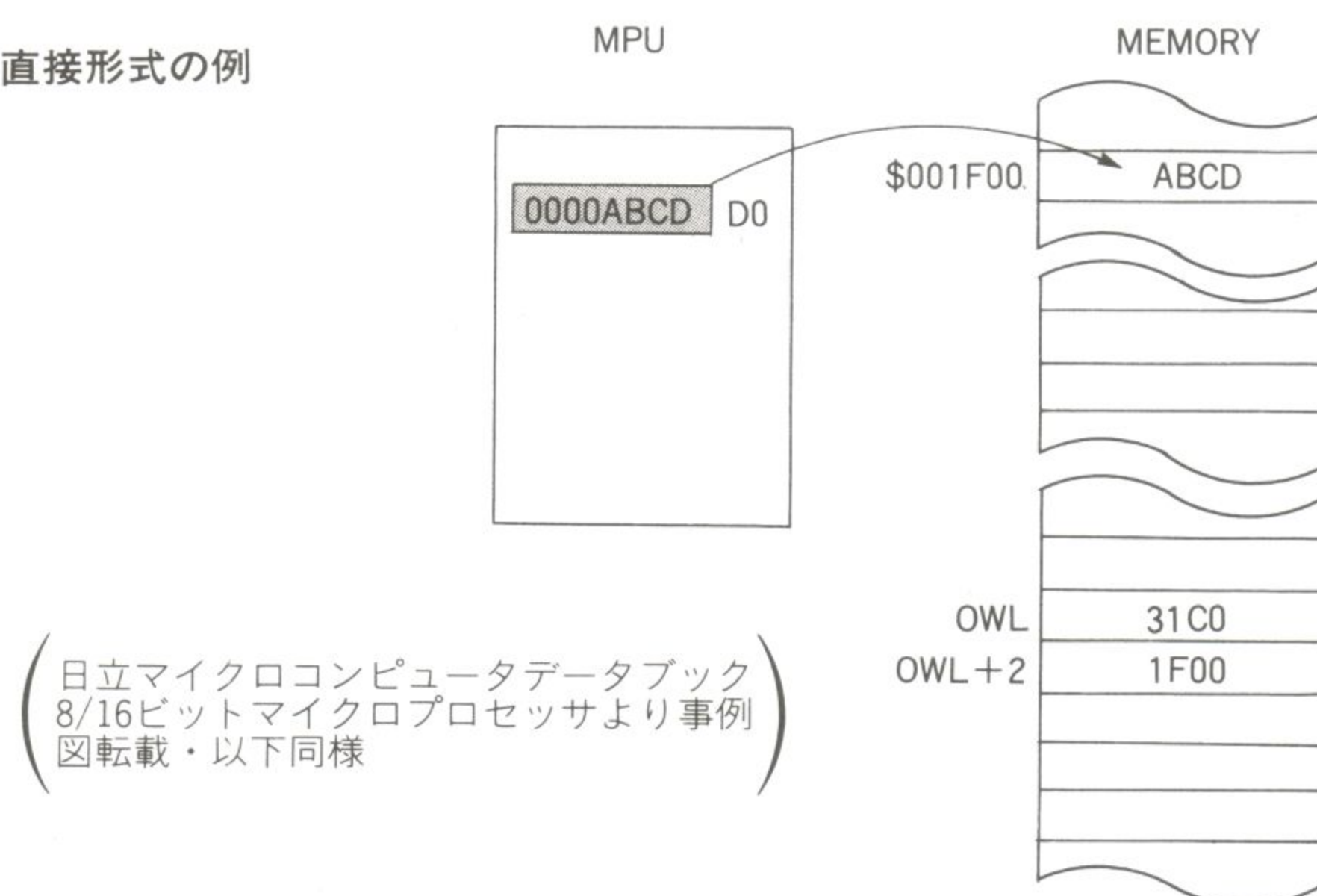
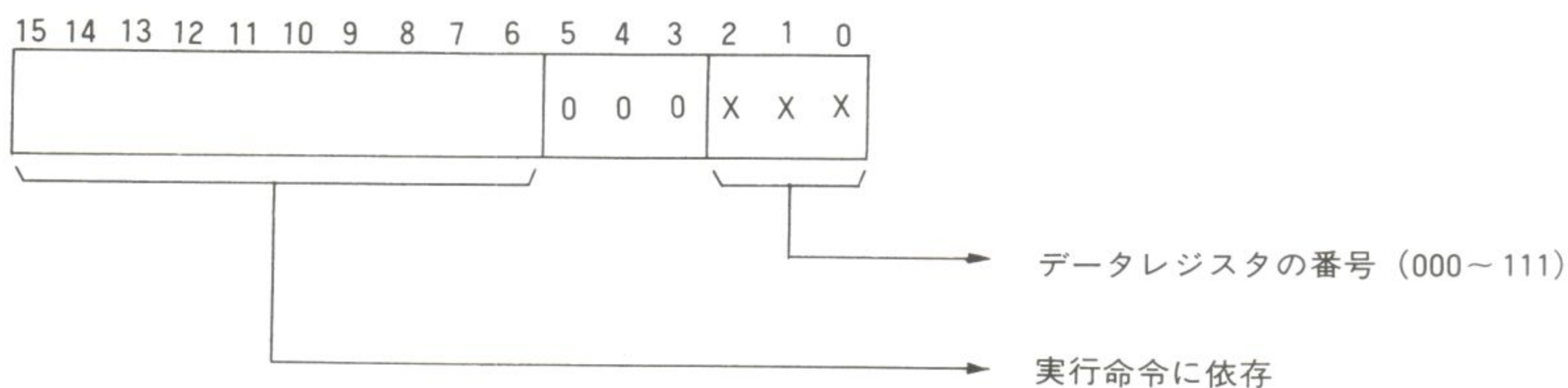
（日立マイクロコンピュータデータブック 8/16ビットマイクロプロセッサより事例図転載・以下同様）

機械語フォーマット

図1.18 データレジスタ直接形式の機械語フォーマット

2 アドレスレジスタ直接形式
(Address register direct)

アセンブラ書式： **An** [n=0〜7]

解説 操作対象となるデータは指定したアドレスレジスタであり,命令はそのレジスタの内容に対して直接実行されます．このモードでは次の2点に注意が必要です．

① バイトサイズはサポートされない．

② アドレスレジスタをディスティネーション・オペランド（結果の格納先）に指定すると，32ビットすべてが対象になり，ソース・オペランドは，符号拡張された32ビットのデータとして扱われる．

MOVE.W A4, $201000

ソース側はアドレスレジスタ直接，ディスティネーション側は絶対ロング形式によるもので，A4の下位ワードがアドレス$201000から連続する2番地へ転送される．

図1.19 アドレスレジスタ直接形式の例●1

MOVE.W $201000, A4

ソース側は絶対ロング，ディスティネーション側はアドレスレジスタ直接によるもので，アドレス$201000から連続する2番地の内容がA4の下位ワードへ転送される．ただしアドレスレジスタは常に32ビットの値を必要とするので，アドレスレジスタへ16ビットの内容を転送したつもりでも，この場合に転送先として指定したA4の上位ワードには，ソースオペランドの符号ビットが満たされる．これを符号拡張といい，「ソースオペランドは符号拡張された32ビットとしてA4へ転送される」と表現される．よって，A4の上位ワードの内容は，$0000または$FFFFのいずれかの値となる．

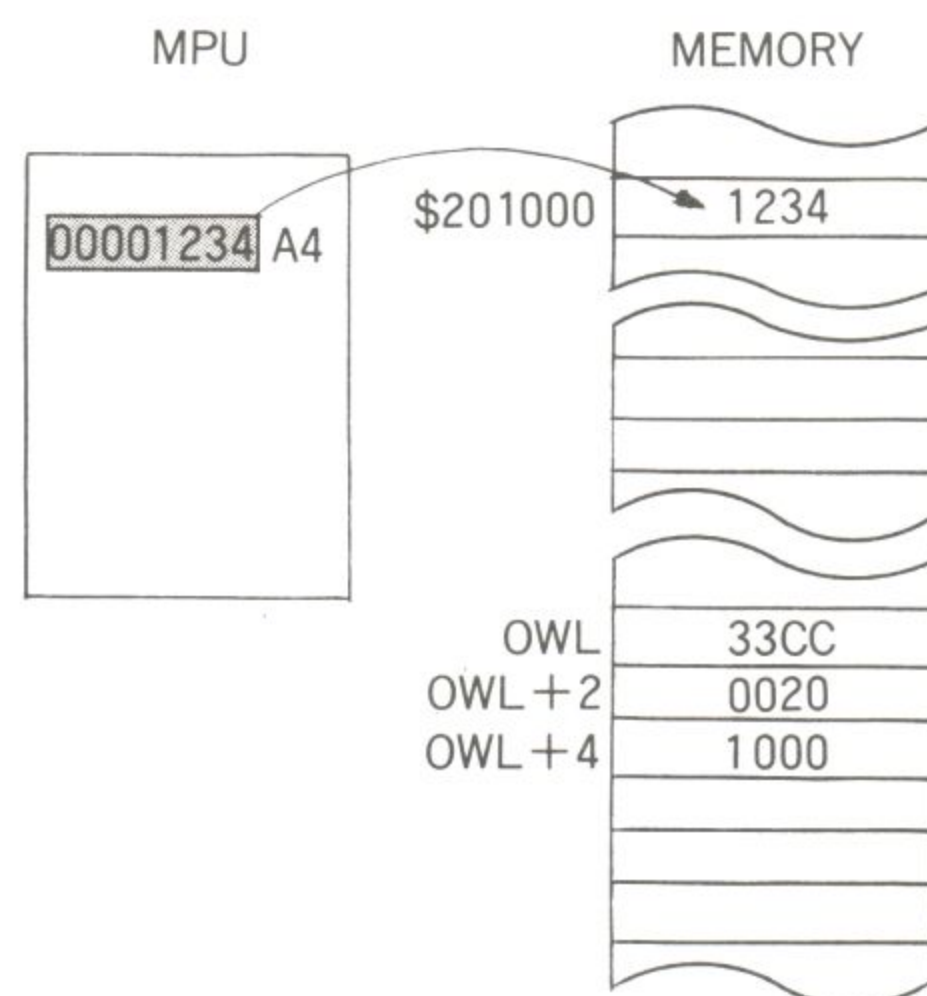



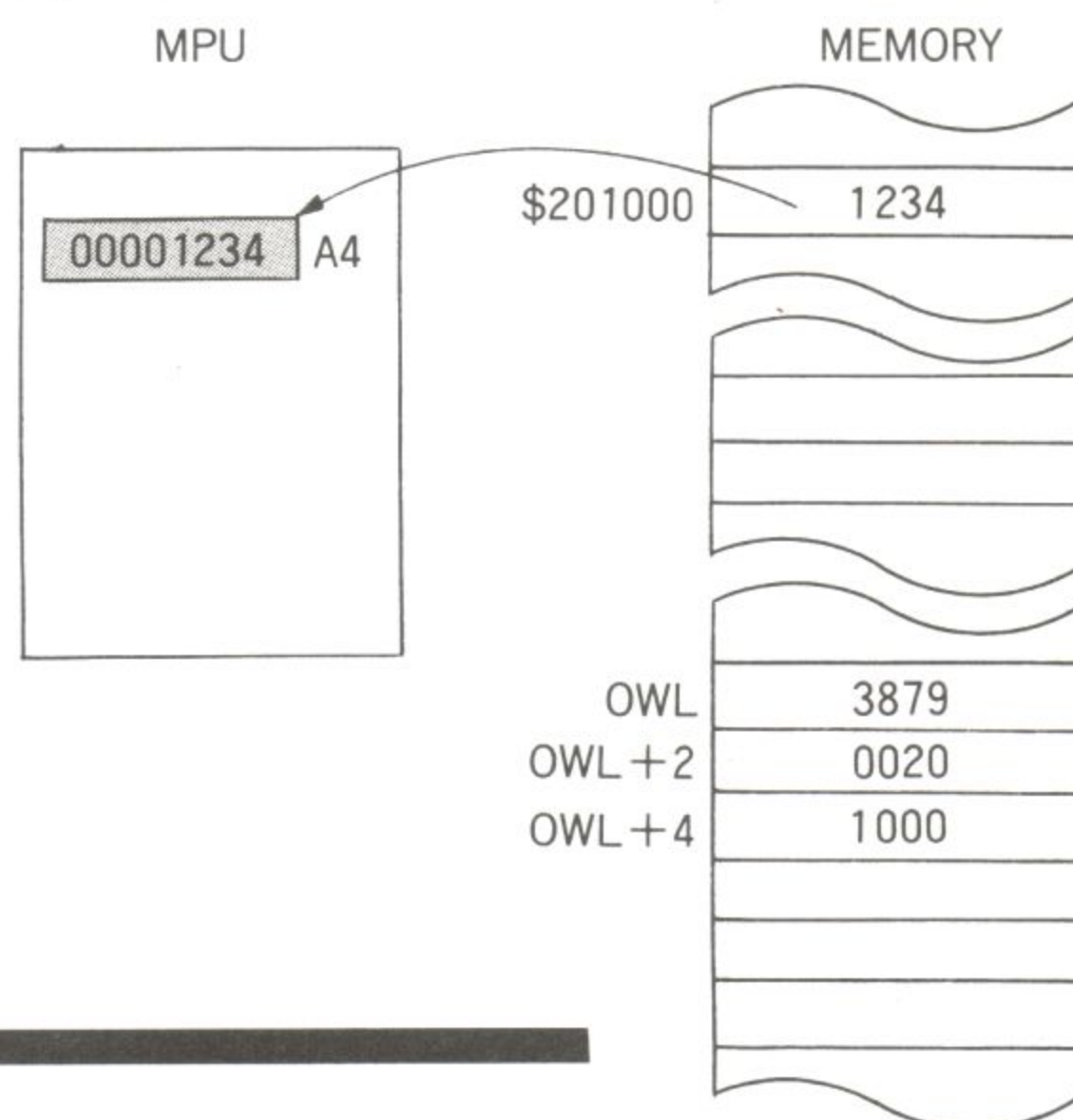
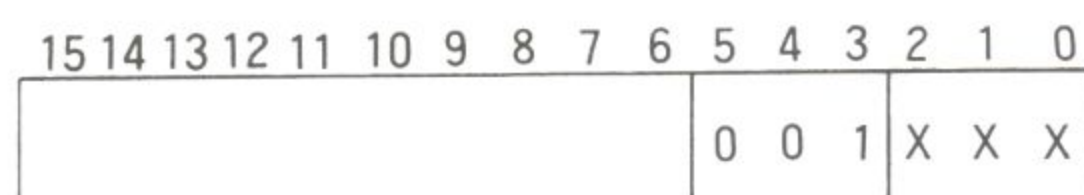
図1.20 アドレスレジスタ直接形式の例●2

機械語フォーマット

図1.21

3 アドレスレジスタ間接形式

(Address register indirect)

アセンブラ書式： (An) [n=0～7]

解説 操作対象となるデータは，指定したアドレスレジスタでポイントされるメモリ上に存在しますが，このような用法としてのアドレスレジスタをポインタと呼び，メモリへのアクセスの基本的なアドレッシングです．

〝(A 0)〟という用法は，アドレスレジスタA 0の内容が操作されるのではなく，A 0の内容はアドレスバスへ出力され，その結果選ばれたメモリの内容が操作の対象となるのです．要するに，

```
MOVE A0, D0
MOVE (A0), D0
```

とは明らかに操作される内容が異なるということです．

レジスタに対しては演算という方法が許され，一度アドレスが決定すれば，そこを起点として前後へ移動できることや，ある間隔をおいた操作など，極めて複雑なメモリ操作が可能になります．極端にいえば，このモードだけでメモリ操作のほとんどを記述できます．

この形式は，JMP，JSRを除きデータ参照に分類されます．

MOVE.W (A0), D0

ソース側はアドレスレジスタ間接，ディスティネーション側はデータレジスタ直接によるもので，A 0でポイントされるメモリから連続した2番地の内容がD 0の下位ワードに転送されるが，D 0の上位ワードは影響を受けず，その内容が失われることはない．

図1.22 アドレスレジスタ間接形式の例

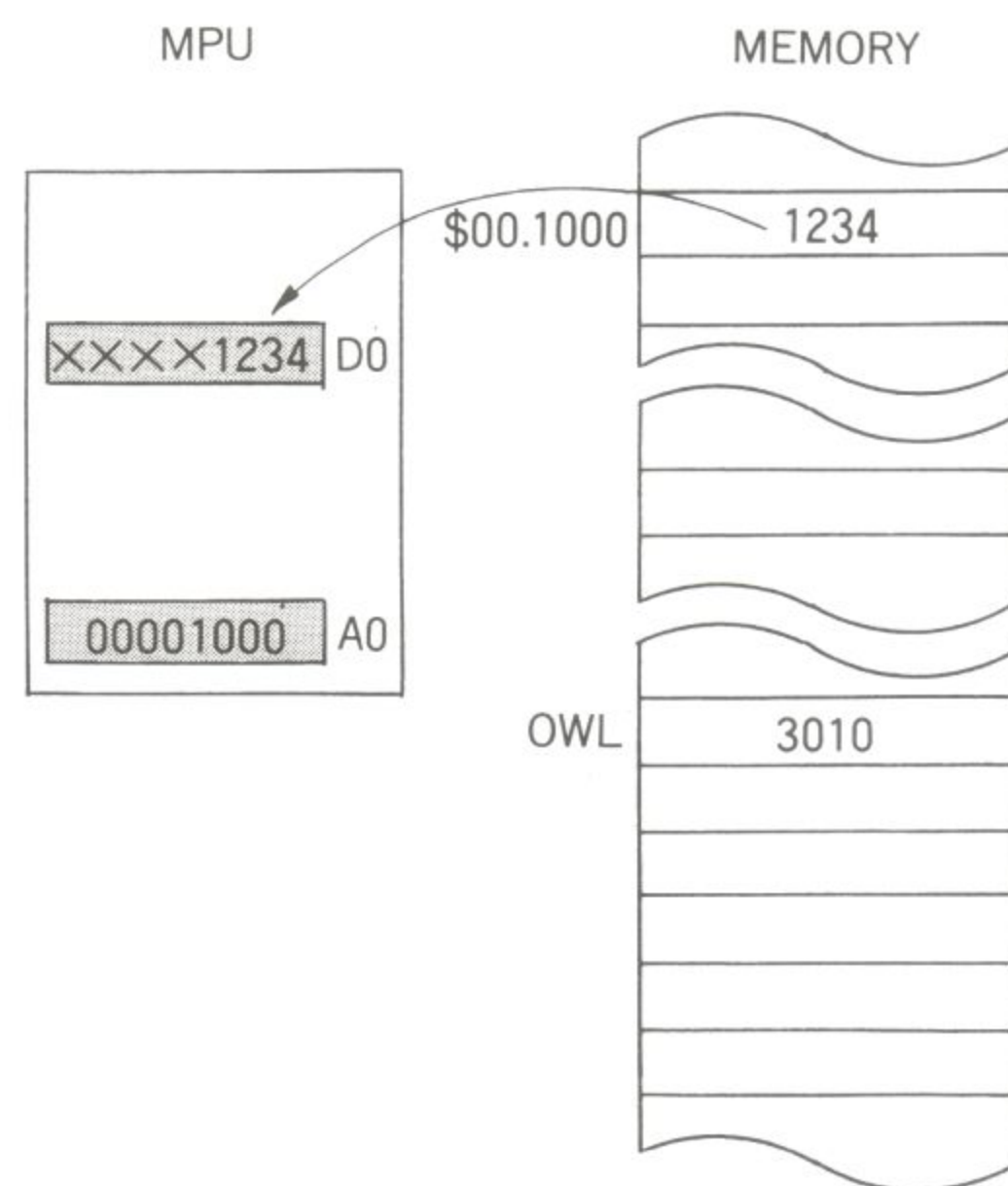
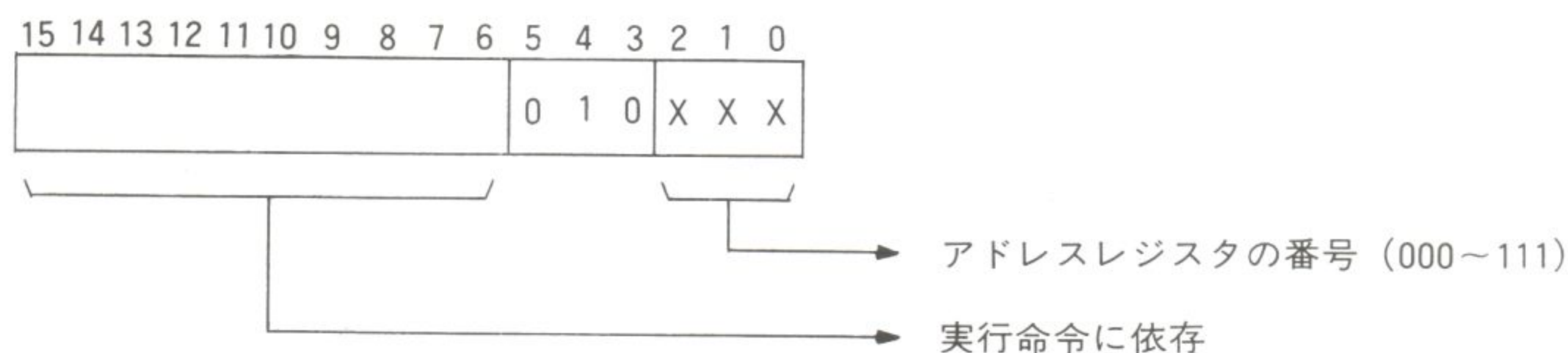
MPU
MEMORY
$00.1000
1234
××××1234 D0
00001000 A0
OWL
3010

機械語フォーマット

図1.23

15 14 13 12 11 10 9 8 7 6	5 4 3	2 1 0
	0 1 0	X X X

X X X → アドレスレジスタの番号（000～111）

15～6 → 実行命令に依存

4 ポストインクリメント・アドレスレジスタ間接形式

(Address register indirect with postincrement)

アセンブラ書式: (An)+ [n=0~7]

解説 操作対象は，指定したアドレスレジスタの内容でポイントされるメモリ上のデータであり，命令実行後,そのアドレスレジスタは,扱ったデータサイズに応じて,バイトなら1が,ワードなら2が,ロングワードなら4が加算されます.

ただし，アドレスレジスタをSPに指定した場合には，バイトデータでも2が加算されます．これは，スタックポインタが偶数値を保持するためです.

システムスタックであるA7あるいはSPでは，ポップ(POP)と呼ばれる操作ですが，その他のレジスタを使用すると，連続している1次元配列への操作を極めて容易に実現できます.

たとえば，4バイトで1組（1要素）であるような配列のデータ構造を連続的に操作したい場合，1組をアクセスした後は自動的にポインタが更新され，即座に次の要素を操作できます．このようなアドレッシングのサポートされていないプロセッサでは，1組のアクセス終了後，ポインタに対する加算命令を実行しなければ，次の要素が格納されているアドレスを生成できません.

この形式は，データ参照に分類されます.

MOVE.W (A4)+, $2000

ソース側はポストインクリメント・アドレスレジスタ間接，ディスティネーション側は絶対ショート形式によるもので，順に以下のような操作となる.

① A4でポイントされるメモリから連続した2番地の内容をアドレス$2000から連続した2番地へ転送する.

② A4はオペレーションサイズに応じてインクリメントされるので,この例では2つだけ増加(A4=A4+2)する.

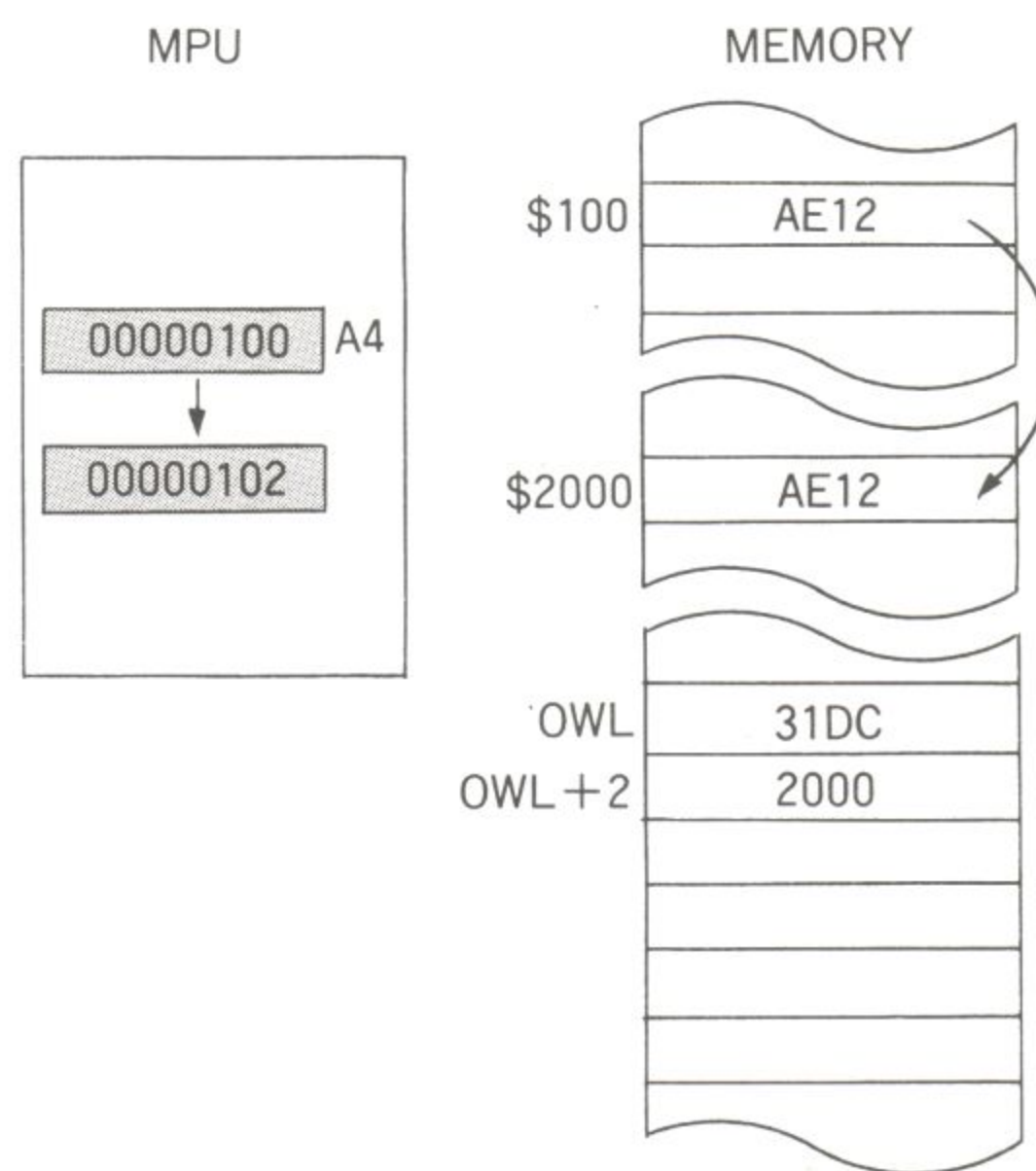
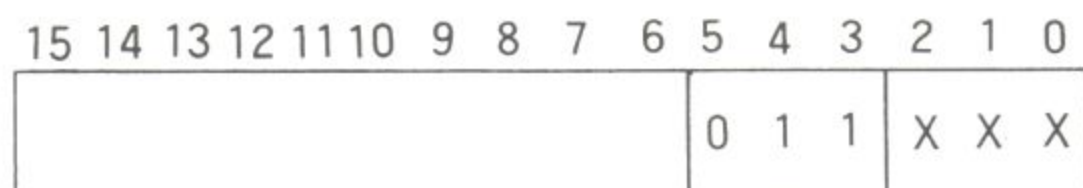
図1.24

機械語フォーマット

図1.25

5 プリデクリメント・アドレスレジスタ間接形式

(Address register indirect with predecrement)

アセンブラ書式： －(An) ［n=0〜7］

解説

命令実行に先ち，アドレスレジスタは，扱うべきデータサイズに応じて，バイトデータなら1，ワードデータなら2，ロングワードデータなら4が減算され，その結果がポイントするメモリ上のデータが命令実行の対象となります（アクセス後，アドレスレジスタの内容は変化しません）．

ただし，〝－ (SP)〟の場合にはバイトデータの操作でも2が減算されます．これはスタックポインタが偶数値でなければならないからです．

システムスタックに対するものは，一般に，PUSHと呼ばれる操作です．

68000には，〝(An)＋〟であれ〝－ (An)〟であれ，すべてのアドレスレジスタに対してサポートされますから，すべてのアドレスレジスタをスタックポインタとして使用でき，このため，PUSHやPOPというスタック操作用の命令が汎用化され，PUSH，POPという命令はありません．これだけでもすばらしいことです．

この形式は，データ参照に分類されます．

MOVE.W －(A3)，＄4000

ソース側はプリデクリメント・アドレスレジタ間接，ディスティネーション側は絶対ショート形式によるもので，順に以下のような操作となる．

① A３はオペランドサイズに応じてデクリメントされるので，ここではA３から２を減じる（A３＝A３－２）

② 先に操作されたA３でポイントされるメモリから連続した２番地の内容が，アドレス＄4000から連続した２番地へ転送される．A３の内容は①で操作された値を保持する．

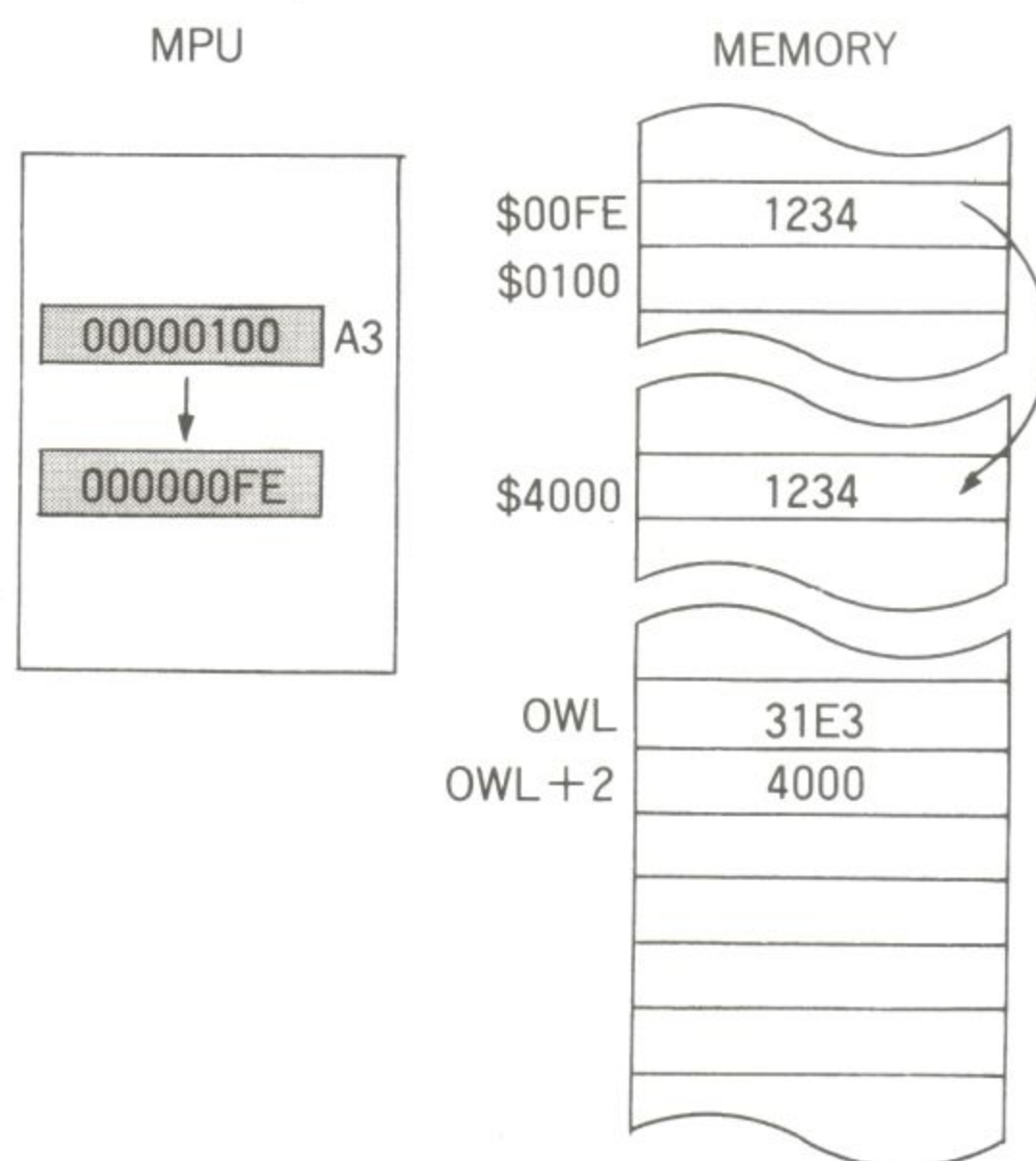
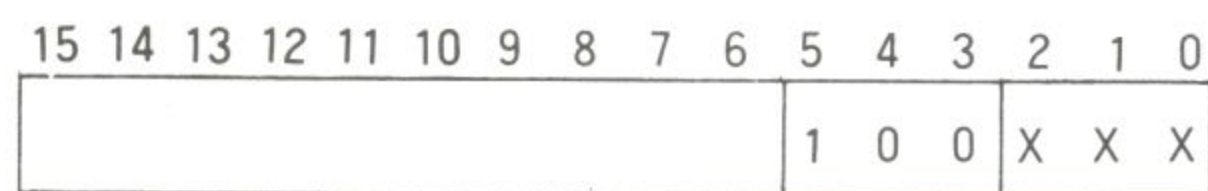
図1.26 プリデクリメント・アドレスレジスタ間接形式の例

機械語フォーマット

図1.27

6 ディスプレースメント付きアドレスレジスタ間接形式

(Address register indirect with displacement)

アセンブラ書式: **d16(An) [n=0~7]**

解説

指定したアドレスレジスタにd 16(−32768〜+32767)のディスプレースメントを加算した内容が,アクセスすべきメモリアドレス(ポインタ)となります.

指定できるディスプレースメントは,16ビットの符号付き整数で表現できる範囲内になければなりませんが,実際にメモリアドレスを計算する場合には,ディスプレースメントの符号拡張を行い,32ビットの数値として扱われます.

この形式では,1ワード長の実効アドレス拡張部を必要とし,指定されたディスプレースメントがそこに16ビットの符号付き整数として格納されます.また,ほとんどのアセンブラではd 16の部分を〈式〉で表現でき,〈式〉は値を持っていますが,絶対値以外にも相対値の記述が許されるアセンブラもあります.

このアドレッシングモードは,BASICでのフィールド文やCコンパイラの構造体と同様なデータ構造が典型的な用途です.たとえば,電話番号を管理したい場合には,住所,氏名,利用者との属性など,個々のデータサイズは一定ではありませんから,先頭から何バイト目に住所フィールドの先頭が記憶される,というようにメモリをフォーマットしておき,ディスプレースメントを指定し,構造化した各要素をアクセスするわけです.

この形式は,JMP,JSRを命令を除き,データ参照に分類されます.

MOVE.W　$100(A0),$3000

ソース側はディスプレースメント付きアドレスレジスタ間接,ディスティネーション側は絶対ショート形式によるもので,ソース側で生成されたアドレスでポイントされるメモリから連続した2番地の内容が,アドレス$3000から連続した2番地へ転送される.

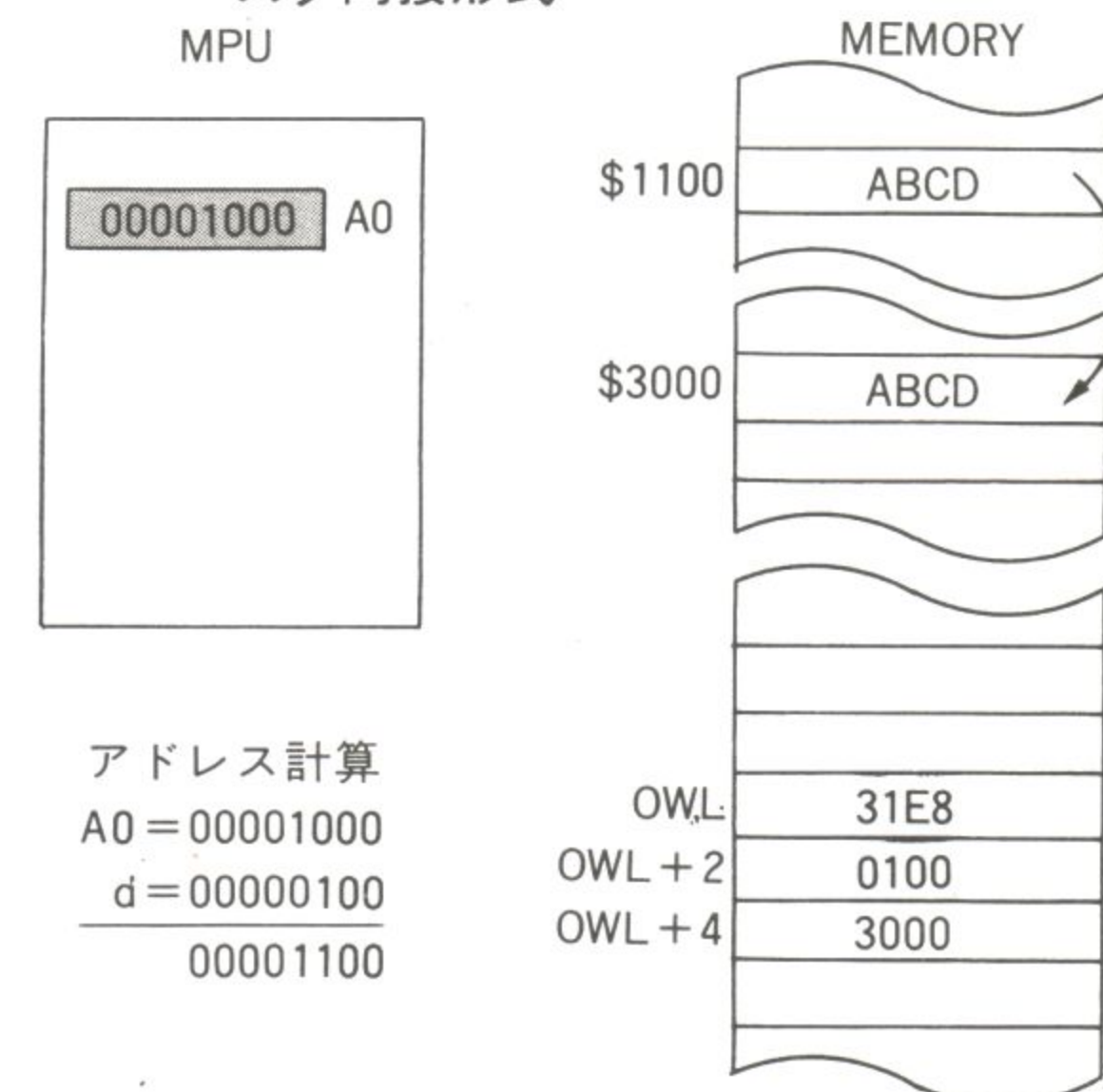
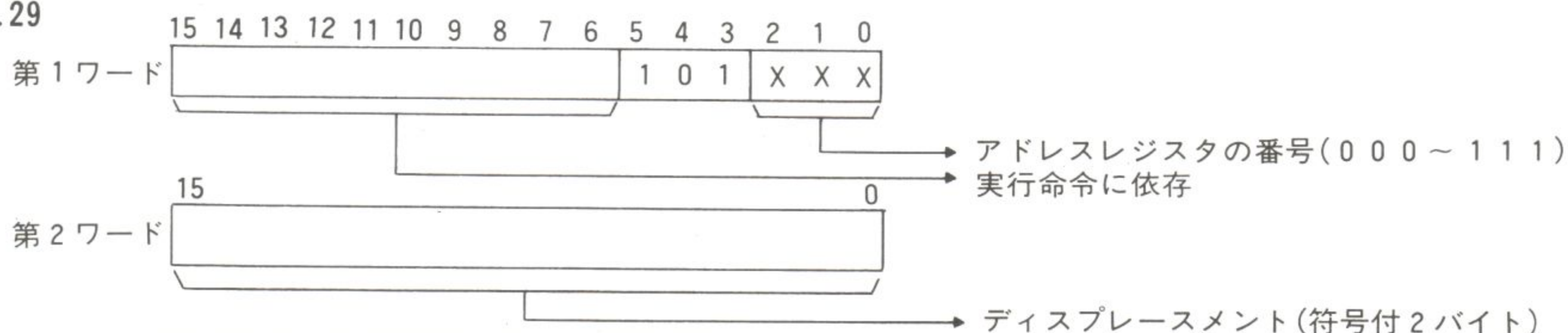
図1.28 ディスプレースメント付アドレスレジスタ間接形式

機械語フォーマット

図1.29

7 インデックス付アドレスレジスタ間接形式

(Address register indirect with index)

アセンブラ書式： **d8(An,IX)** [IX=An or Dn (n=0〜7)]

解説

アドレスレジスタの内容にインデックスレジスタIXとディスプレースメントd 8 (−128〜+127) を加えたメモリアドレス上にあるデータが，命令実行の対象になりますが，d 8は，32ビットに符号拡張されてアドレスの計算に用いられます．

この形式では，1ワード長の実効アドレス拡張部が必要で，この拡張ワード内にインデックスレジスタやディスプレースメントに関する情報が格納されます．またほとんどのアセンブラでは，d 8の部分に〈式〉を記述できますが，これは絶対値でなければなりません．

このアドレッシングモードは，68000のサポートする16Mバイトの全アクセス空間を有効に活用するような応用，たとえば，より複雑なデータ構造，多次元配列などの操作，が典型的な用途です．そこでは，Anにベースアドレスを IXにはKeyをセットして目的の場所をアクセスします．IXはその名の通り索引（インデックス）として，目的場所への指針，指標となるわけです．

この形式は，JMP，JSR命令を除き，データ参照に分類されます．

ディスプレースメント，IXレジスタについての要点は次のようになります．

① ディスプレースメントは8ビットの符号付整数で、−128〜+127の範囲である．

② インデックスレジスタIXには，アドレスレジスタ以外にもデータレジスタを指定でき，〝.W〞，〝.L〞を付加してインデックスレジスタの下位ワードを使用するのか，すべて（ロングワード）を使用するかを指定するが，省略すると〝.W〞が指定されたものと解釈される．

③ 実効アドレスの算出はすべてロングワードで行なわれ，指定されたディスプレースメントのみならず，〝IX.W〞では，指定したインデックスレジスタの内容を符号付き16ビット整数と解釈し，32ビットに符号拡張した数値としてアドレスが求められ，〝IX.L〞なら32ビットすべてが用いられる．

MOVE.W　＄04(A0，D0)，＄1000

ソース側はインデックス付アドレスレジスタ間接（D０がインデックスレジスタ），ディスティネーション側は絶対ショート形式によるもので，ソース側で生成されたアドレスでポイントされるメモリから連続した２番地の内容が，アドレス＄1000から連続した２番地へ転送される．

図1.30　インデックス付アドレスレジスタ間接形式の例

MPU
00002BDC D0
00002000 A0

MEMORY
$1000 2345
$4BE0 2345
OWL 31F0
OWL＋2 0004
OWL＋4 1000

アドレス計算
A0＝00002000
D0＝00002BDC
d＝00000004
00004BE0

機械語フォーマット

図1.31

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
第１ワード | 110 | XXX
XXX → アドレスレジスタの番号
15〜6 → 実行命令に依存

15 14 13 12 11 10 9 8 7 0
第２ワード R i i i S 0 0 0 |
ディスプレースメント（符号付１バイト）
インデックスレジスタのサイズ
0：ワードデータとして扱う
（符号拡張した32ビットとしてアドレスの計算に使用される）
1：ロングワードデータとして扱う
インデックスレジスタの番号（000〜111）
インデックスレジスタの種類
0：データレジスタ
1：アドレスレジスタ

8 絶対ショートアドレス形式

(Absolute short address)

アセンブラ書式：〈2バイトの絶対アドレス〉

解説 実行対象であるデータが格納されているアドレスを，2バイトの絶対アドレスで直接指定するモードです．

機械語フォーマットですが，絶対ショート形式では，第2ワードに16ビットの絶対アドレスが格納され，第3ワードは作成されませんが，実際にアクセスされるアドレスは，符号拡張された32ビットの値が採用されます．

使用前に符号拡張された結果，アクセス可能なアドレスは**表1.10**のようになります．

表1.10 アクセス可能なアドレス

指定範囲	アクセス範囲
$0000～$7FFF $8000～$FFFF	$00000000～$00007FFF $FFFF8000～$FFFFFFFF

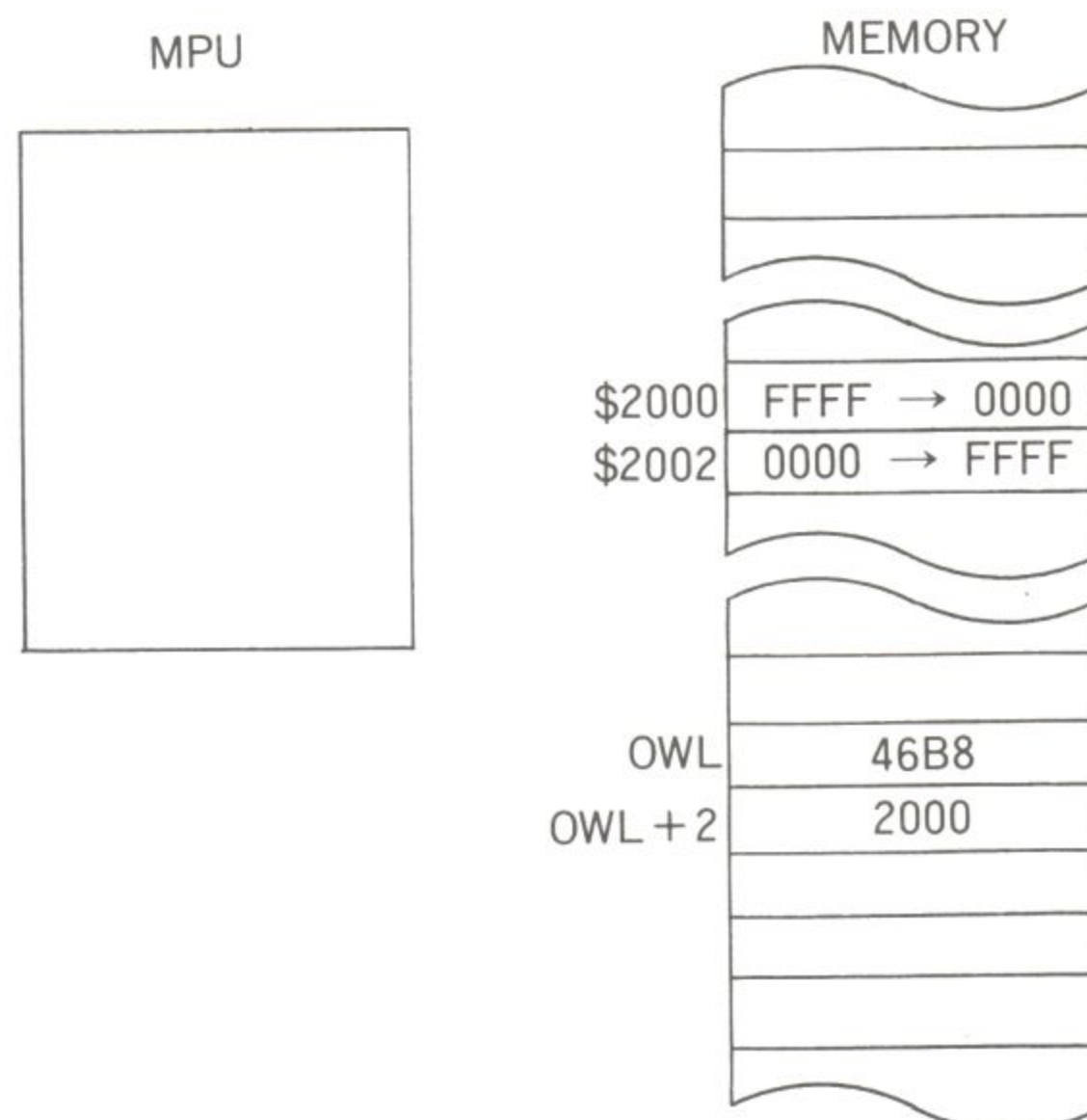
したがって，$00008000～$FFFF7FFFまでの範囲を絶対ショート形式で参照することはできません．また，どのように解釈するかという問題はアセンブラの文法に依存し，2バイトの絶対アドレスを指定しても，常に$0000を上位に補って，絶対ロングのマシンコードへ変換するアセンブラや，$0000～$7FFFまでを絶対ショート形式と解釈するアセンブラもあるようですから，簡単なアセンブルテストで確認すべきでしょう．

通常は，絶対アドレス形式が使用されるケースは稀ですが，最下位32Kバイトには，MPUのベクタ領域が位置し，OSにとっても重要な作業用エリアなども存在するでしょう．さらに，I／O空間を最上位32Kバイトに割り当てると，1ワードで16Mバイトの全アドレス空間内の最下位32Kバイトと最上位32Kバイトの範囲をアクセスできるので，これらの領域を絶対ショート形式で参照できます．

この形式は，JMP，JSR命令を除き，データ参照に分類されます．

NOT.L　$2000

オペランドは絶対ショートアドレス形式であり，アドレス$2000から連続する4番地の内容に対してNOT演算（否定をとる）を行う．

図1.32　絶対ショートアドレス形式の例1

MOVE.W　$1000，$2000

ソース，ディスティネーションともに絶対ショートアドレス形式であり，$1000から連続する2番地の内容が，$2000から連続する2番地へ転送される．

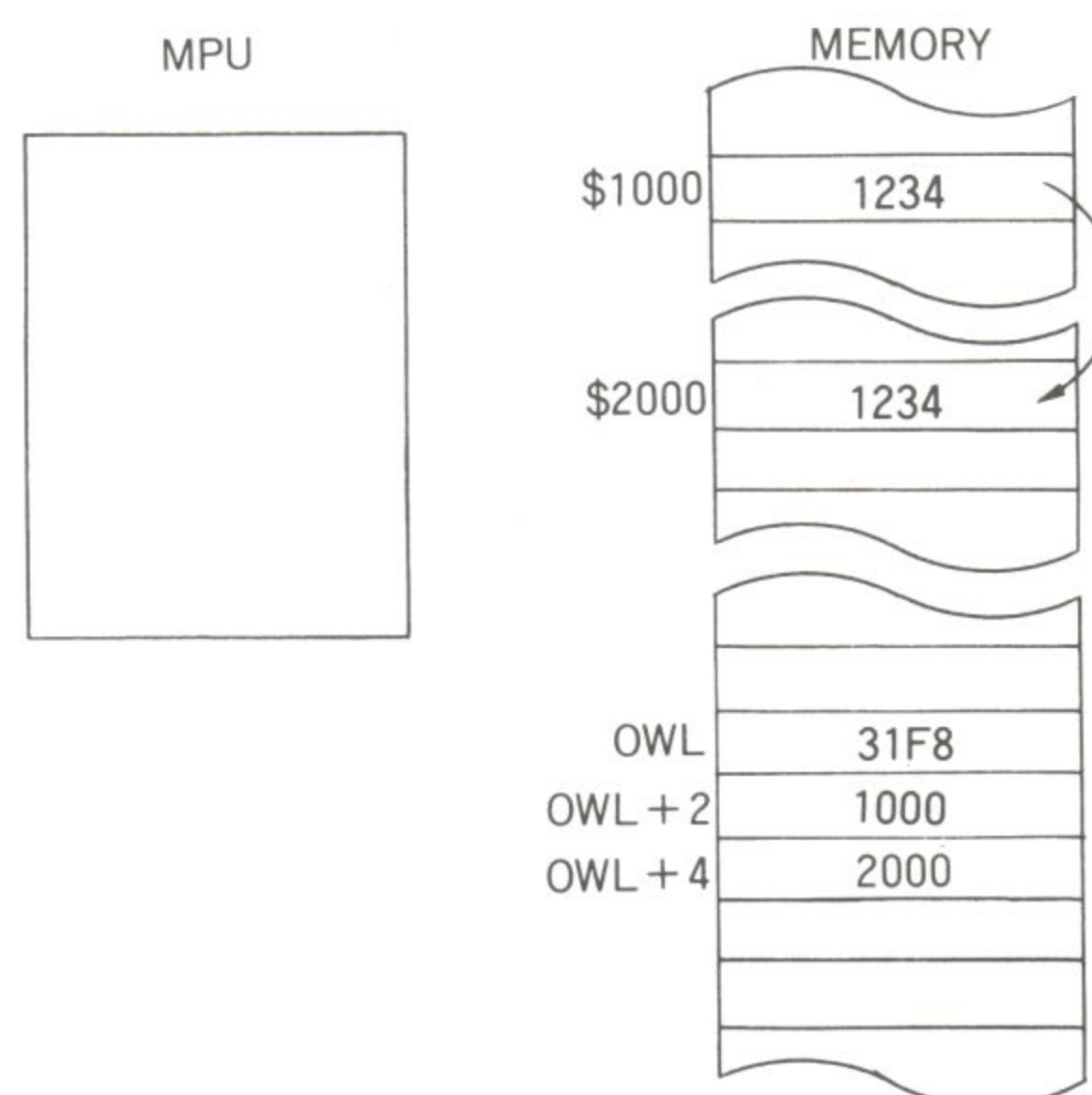
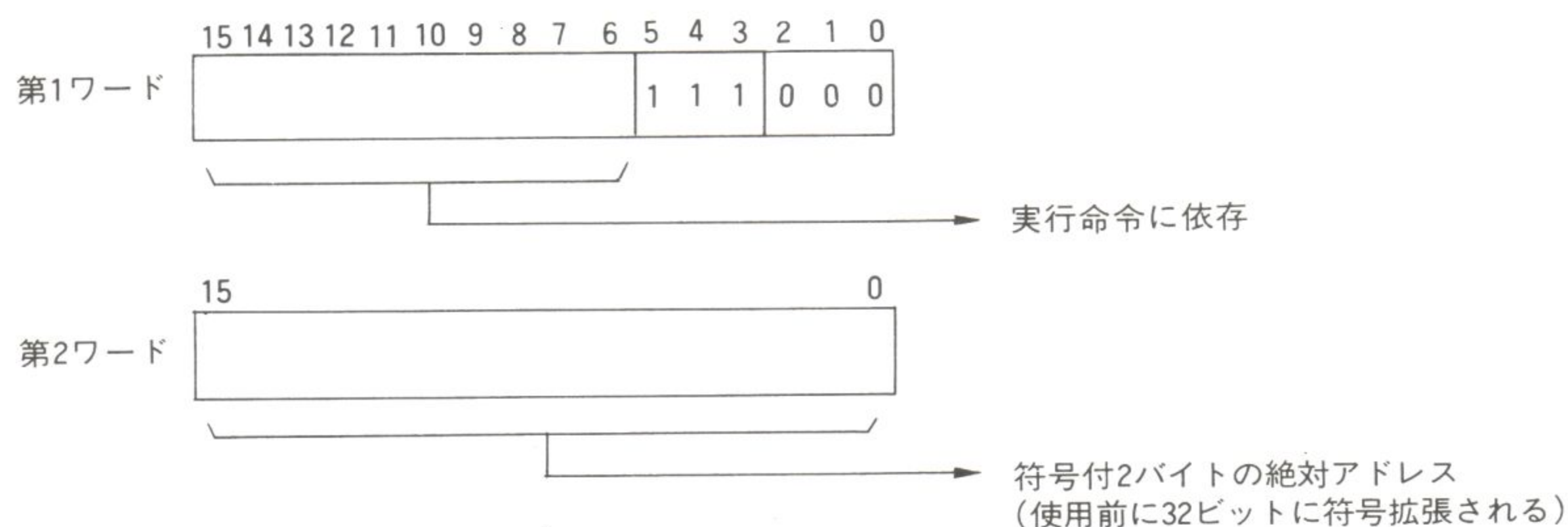
図1.33　絶対ショートアドレス形式の例2

機械語フォーマット

図1.34

9 絶対ロングアドレス形式
(Absolute long address)

アセンブラ書式：〈2バイト以上の絶対アドレス〉

解説 実行対象であるデータが格納されているアドレスを，2バイト以上の絶対アドレスで直接指定するモードです．

絶対ロング形式では，第2ワードに32ビット絶対アドレスの上位16ビットが，第3ワードに下位16ビットが格納されます．

この形式は，JMP，JSR命令を除き，データ参照に分類されます．

NEG.W　$014000

オペランドは絶対ロングアドレス形式であり，メモリ$14000から連続する2番地の内容の符号を反転する．$FFFFとはマイナス1（−1）のことである．

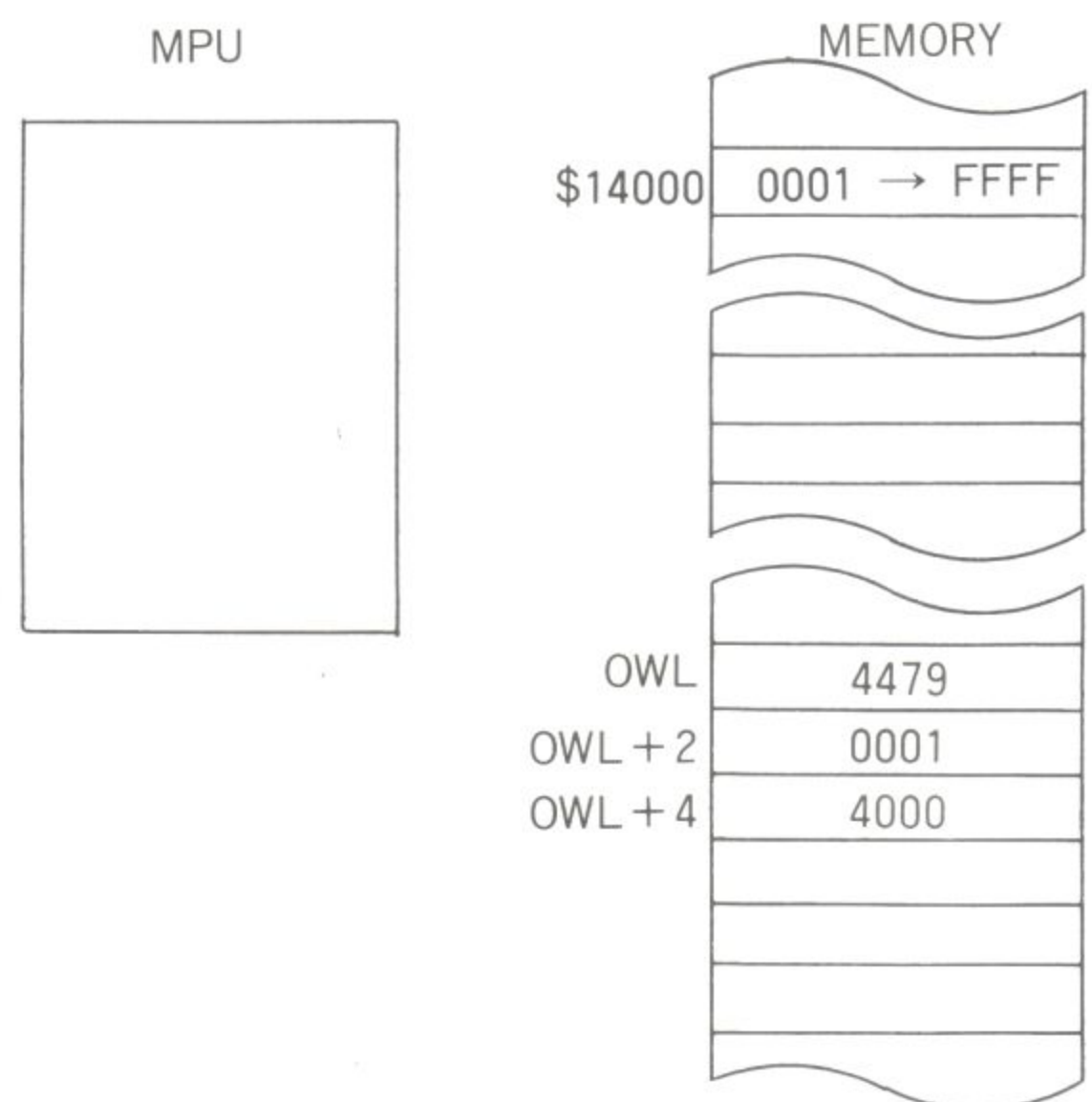
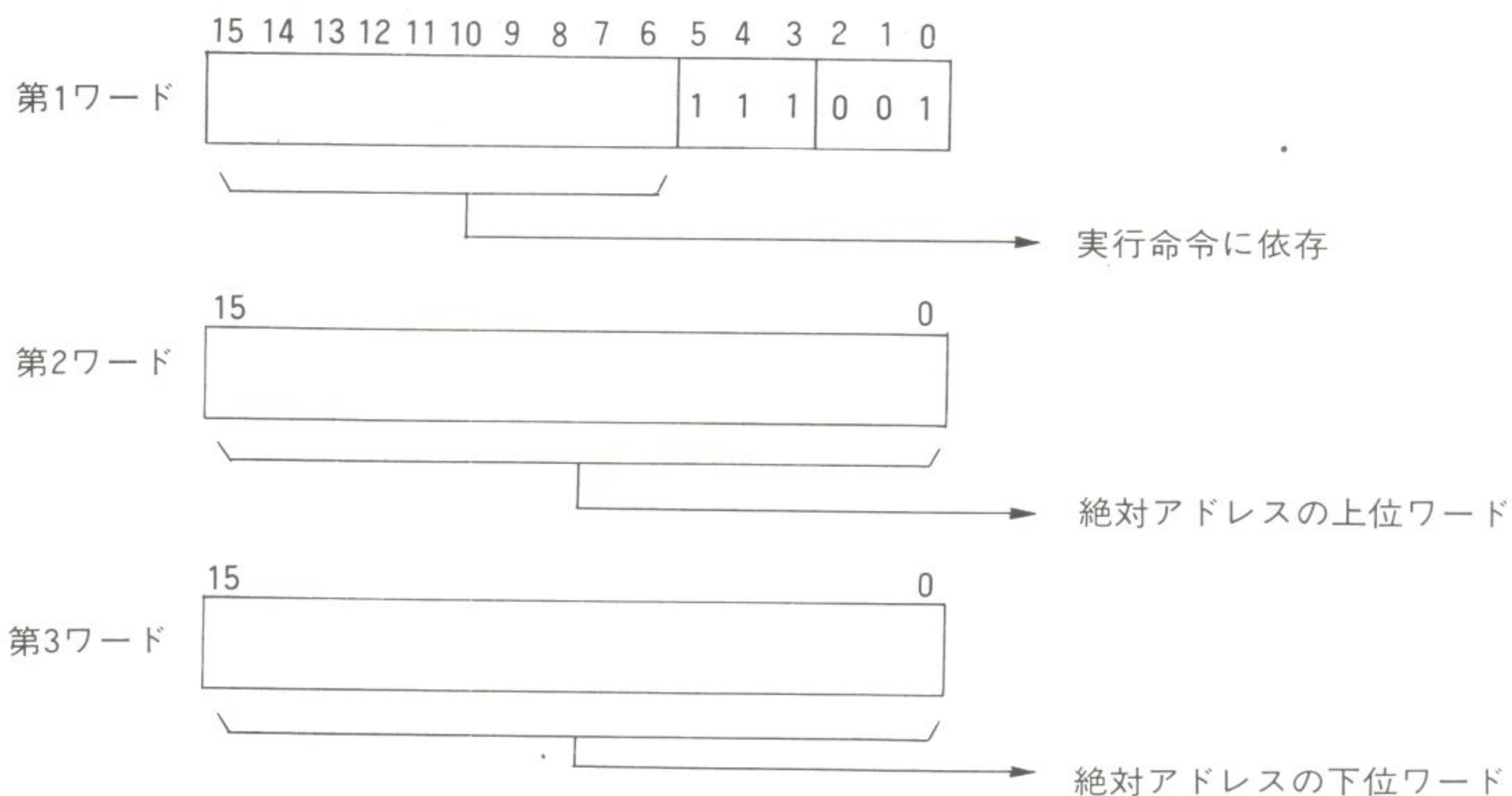
図1.35　絶対ロングアドレス形式の例

機械語フォーマット

図1.36

10 ディスプレースメント付プログラムカウンタ相対形式

(Program counter with displacement)

アセンブラ書式： **d16(PC)**

解説

オペランドのアドレスは，プログラムカウンタとディスプレースメントの総和であり，命令実行の対象となるデータのシンボル名，またはその命令からの相対的な距離を指定します．

ディスプレースメントの範囲は－32768～＋32767ですが，プログラムカウンタの値は，命令の先頭アドレスから＋2進んだ地点の拡張ワード部をポイントしており，ここから指定シンボルまでのディスプレースメントを求めていることになり，命令の先頭アドレスからのディスプレースメントならば，－32766～＋32769となります．

ディスプレースメントに絶対値を指定するケースは稀であり，多くの場合，分岐命令とともに使用され，d16には分岐先のシンボル（ラベル）を〈式〉として記述できます．つまり，現在のプログラムカウンタの値からシンボル（ラベル）までの相対的な距離をアセンブラが算出し，機械語フォーマットの第2ワードである実効アドレス拡張部へ，ディスプレースメントとしてセットします．

この形式は，プログラム参照に分類され，命令実行の結果，値が更新されるようなデータに対しては指定できません．たとえば，MOVE命令やADD命令の第2オペランド，CLR命令のオペランドなどです．

APPENDIX：リロケータブルという意味について

ここでいうリロケータブルなプログラムとは，プログラムをロードした後でも，つまり，どのうよなロケーションへロードされようとも，動作するプログラムであり，ロード時にアドレスの計算さえ必要としないプログラムを意味します．

通常の走行環境では，プログラムのロードアドレスは開発段階に決定しているか，そうでない場合には，ロード時にアドレスが計算されます．この意味では，リロケータブルでなくともよいのですが，あるアプリケーションでは，同一システム内の異なったアドレスでも走行させたいことがあります．

たとえば，デバッガという「プログラムをデバッグするシステム」は，ターゲットとなるプログラムと重複したのでは都合が悪いわけで，システム（デバッガ）自体をターゲット・プログラムとは別の場所へ転送しなければなりません．もし，デバッガ自体がその場所で走行できなければ，デバッガの説明書には，「デバッガと同一のメモリ領域で動作するプログラムはデバッグできません」と記載されていることでしょう．

以上のような要求を満たすプログラムは，リロケータブル・アセンブラで開発すれば解決できるわけではありません．リロケータブル・アセンブラでも，最終的には，特定のアドレスで走行するようなオブジェクト・コードを生成するからです．

MOVE.W　LABEL（PC），D0

ソース側はディスプレースメント付きプログラムカウンタ相対形式，ディスティネーション側はデータレジスタ直接形式で，実効アドレスの計算には，現在のPC，ディスプレースメント，が使用される．

ソース側で生成されたアドレスでポイントされるメモリから連続した2番地の内容が，D0の下位ワードへ転送される．

① PCの値は本命令の実効アドレス拡張部をポイントしているので＄8002になっている．
② ディスプレースメントはアセンブラの文法上，〝LABEL〟というような記号表記が可能であり，D0へ転送すべき内容が格納されているアドレスを，あたかも直接指定するような記述が可能である．
③ ディスプレースメントは符号拡張される．
④ 本例ではディスプレースメントが＄1000ということになっているが，これらの値は，アセンブラが算出するので，プログラマが関知する必要はないし，そのためにアセンブラという機械語変換プログラムの意義がある．

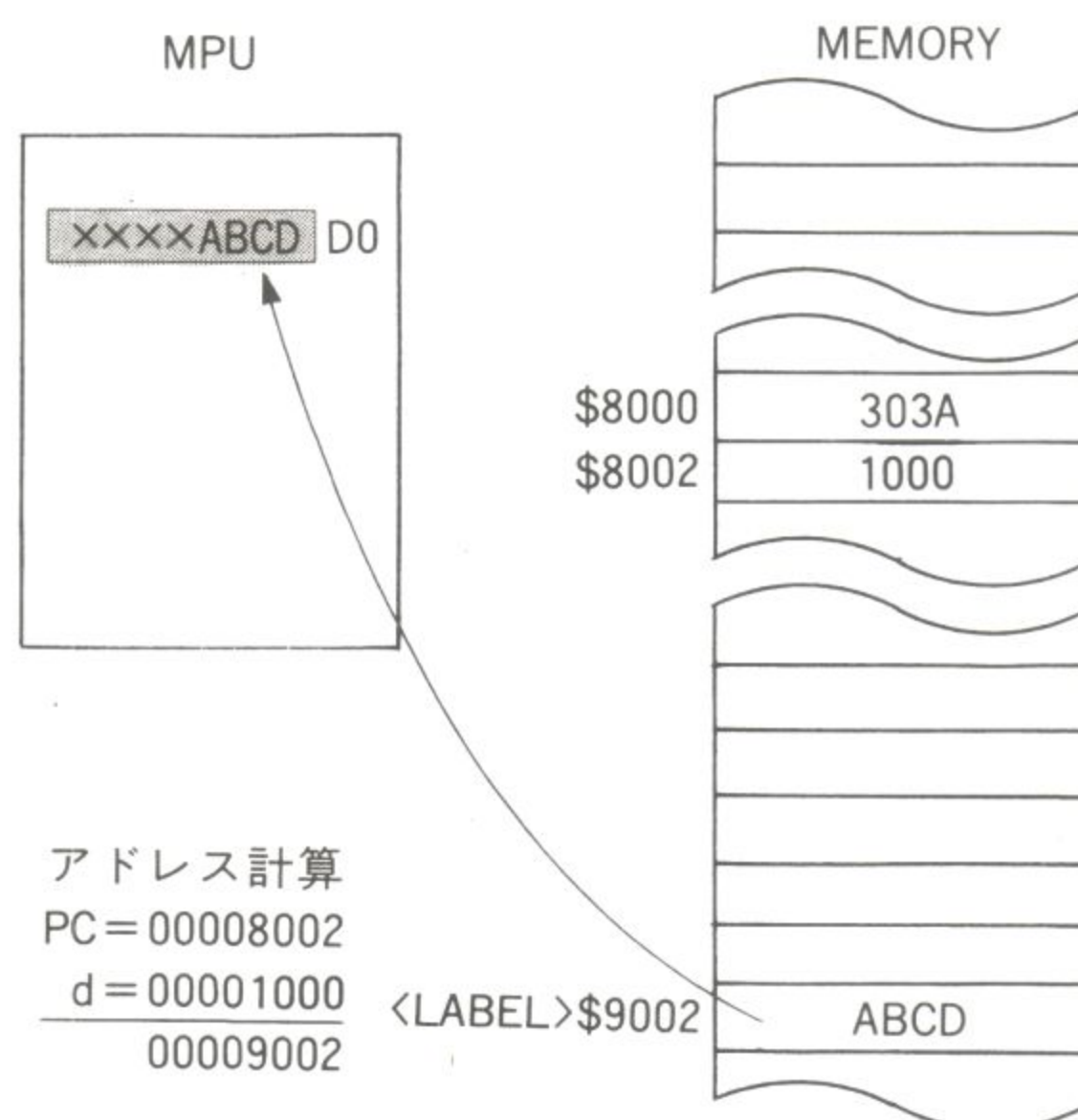
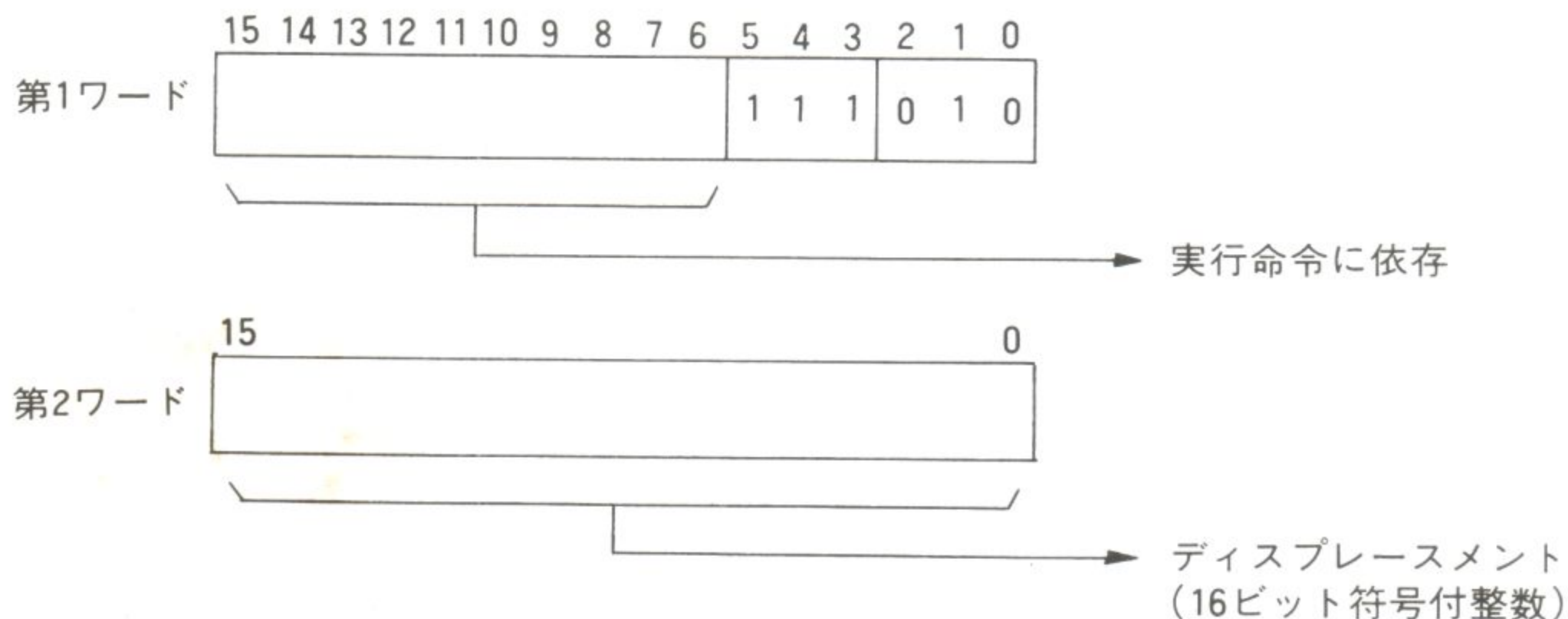
図1.37　ディスプレースメント付プログラムカウンタ相対形式の例

機械語フォーマット

図1.38

11 インデックス付プログラムカウンタ相対形式

(Program counter with index)

アセンブラ書式： **d8(PC,IX)** ［IX=An or Dn (n=0〜7)］

解説 オペランドのアドレスは，プログラムカウンタとインデックスレジスタとディスプレースメントの総和であり，命令実行の対象となるデータの相対シンボル（ラベル）とインデックスレジスタを指定します．

d8は8ビットのディスプレースメントであり，現在のプログラムカウンタの値からの相対距離が−128〜127の範囲に限定されます．現在のプログラムカウンタの値は，その命令の実効アドレス拡張部（第2ワード）をポイントしています．

ディスプレースメント付きプログラムカウンタ相対形式では，符号付き2バイトの範囲がリロケータブルの対象でしたが，この形式は，8本のアドレスレジスタ，8本のデータレジスタをインデックスレジスタに使用でき，これらのサイズは32ビットであるので，68000のサポートする全域を，リロケータブルなコード範囲としてカバーできることになります．

この形式は，プログラム参照に分類されます．

ディスプレースメント，IXレジスタについての要点は次のようになります．

① ディスプレースメントは8ビットの符号付整数で，−128〜＋127の範囲である．

② インデックスレジスタIXには，アドレスレジスタ以外にもデータレジスタを指定でき，〝.W〞，〝.L〞を付加してインデックスレジスタの下位ワードを使用するのか，すべて（ロングワード）を使用するかを指定するが，省略すると〝.W〞が指定されたものと解釈される．

③ 実効アドレスの算出はすべてロングワードで行われ，指定されたディスプレースメントのみならず，〝IX.W〞では，指定したインデックスレジスタの内容を符号付き16ビット整数と解釈し，32ビットに符号拡張した数値としてアドレスが求められ，〝IX.L〞なら32ビットすべてが用いられる．

MOVE.W　LABEL (PC, A0), D0

ソース側はインデックス付プログラムカウンタ相対形式，ディスティネーション側はデータレジスタ直接形式で，実効アドレスの計算には，現在のPC，ディスプレースメント，それに，インデックスとして指定したＡ０が使用される．

ソース側で生成されたアドレスでポイントされるメモリから連続した２番地の内容が，Ｄ０の下位ワードへ転送される．

① PCの値は本命令の実行アドレス拡張部をポイントしているので，＄8002になっている．
② Ａ０の下位ワードを指定しているので，その値は＄1010だが，実効アドレス計算時に符号拡張された32ビットとなる(Ａ０はIXである)．
③ ディスプレースメントはアセンブラの文法上，〝LABEL〟というような記号表記が可能であり，D0へ転送すべき内容が格納されているアドレスをあたかも直接指定するような記述が可能である．
④ ディスプレースメントは符号拡張される．
⑤ 本例ではディスプレースメントが＄10ということになっているが，これらの値は，アセンブラが算出するので，プログラマが関知する必要はないし，そのためにアセンブラという機械語変換プログラムの意義がある．

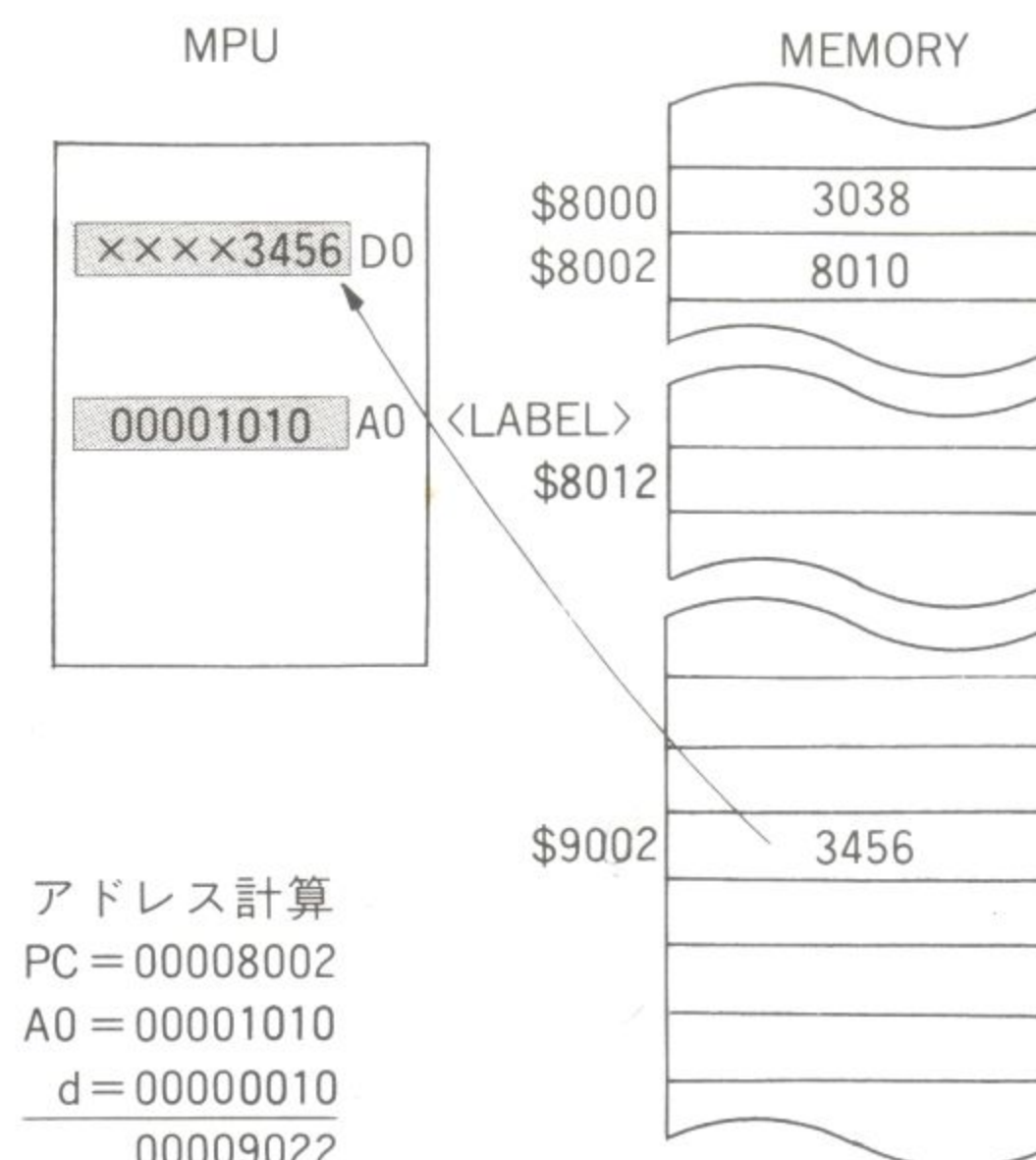
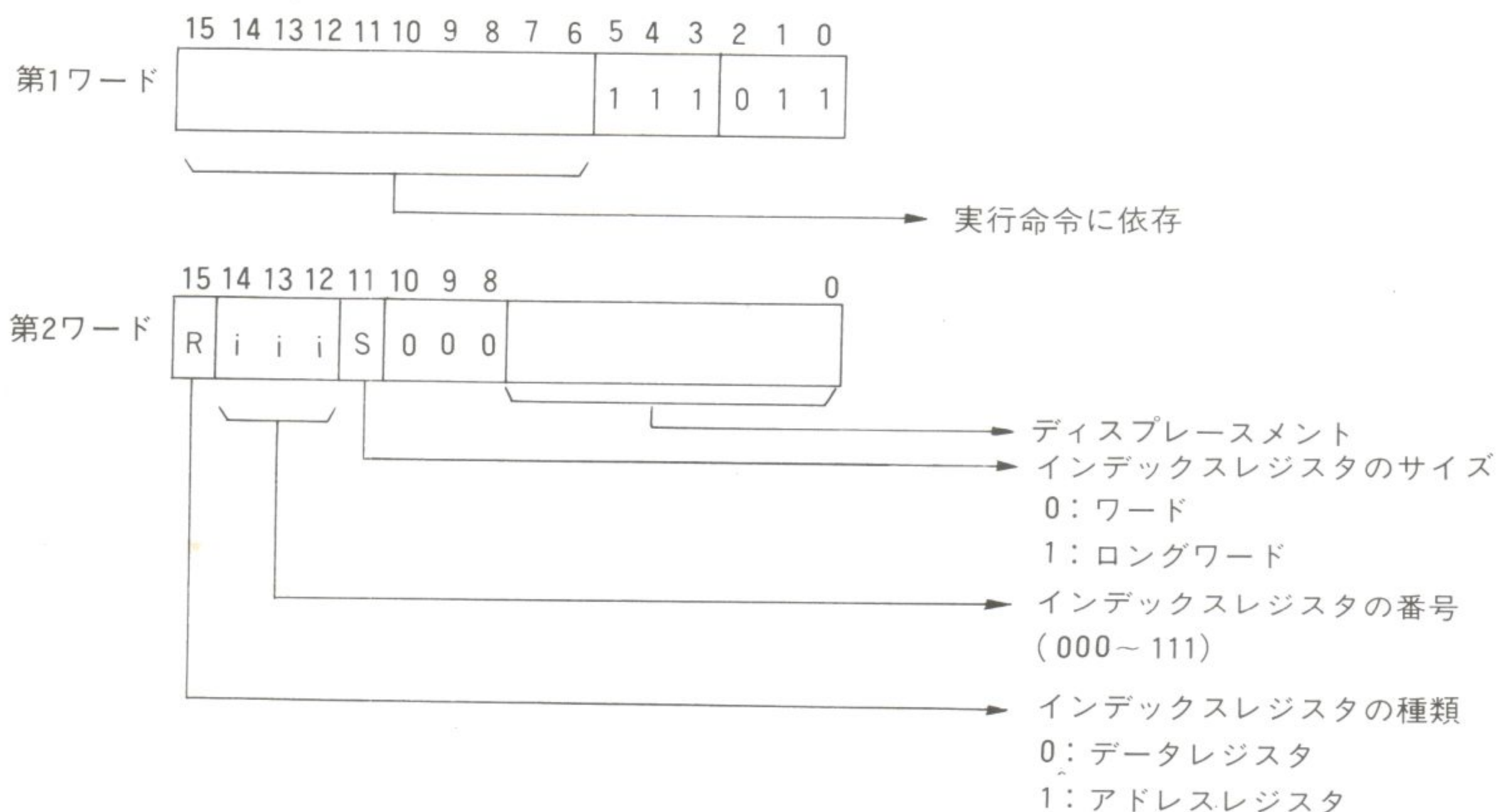
図1.39　インデックス付プログラムカウンタ相対形式の例

機械語フォーマット

図1.40

12 イミディエイトデータ形式
(Immediate data)

アセンブラ書式： #〈絶対値〉

解説 対象となるデータを直接指定するモードで，"#" を先頭に付けることによって，その後に位置する値がデータそのもの（単なる値）であることを示し，ディスプレースメントや絶対アドレスを意味する値とは区別されます．

アセンブラでは〈絶対値〉の部分に式を記述できますが，イミディエイト形式であることを意味する "#" を忘れてはなりません．そこで，16進イミディエート値なら "#$1000" のように，"#" と "$" が連続することになります．

この形式の機械語フォーマットですが，指定されたイミディエイト値のサイズにより，1ないし2ワードの拡張部が付加されますが，扱うデータサイズが，指定したイミディエイトデータのサイズより小さい場合，つまり，イミディエイトデータ・フィールドへ格納できないイミディエイト値は指定できません．

① バイトあるいはワードのイミディエイト値であれば，第2ワードにイミディエイト値がセットされ，第3ワードは作成されない．

② ロングワード(32ビット)のイミディエイト値が指定された場合，第2ワードに上位16ビットが，第3ワードに下位16ビットがセットされる．

MOVE.W #$1000, A0

ソース側はイミディエイトデータ形式，ディスティネーション側はアドレスレジスタ直接形式によるもので，$1000（10進の4096）がA0へ転送される．ただし，アドレスレジスタへ値をワードサイズで格納する場合，68000では常に32ビットのアドレスを必要とするので，符号拡張された32ビットがA0へ格納される．

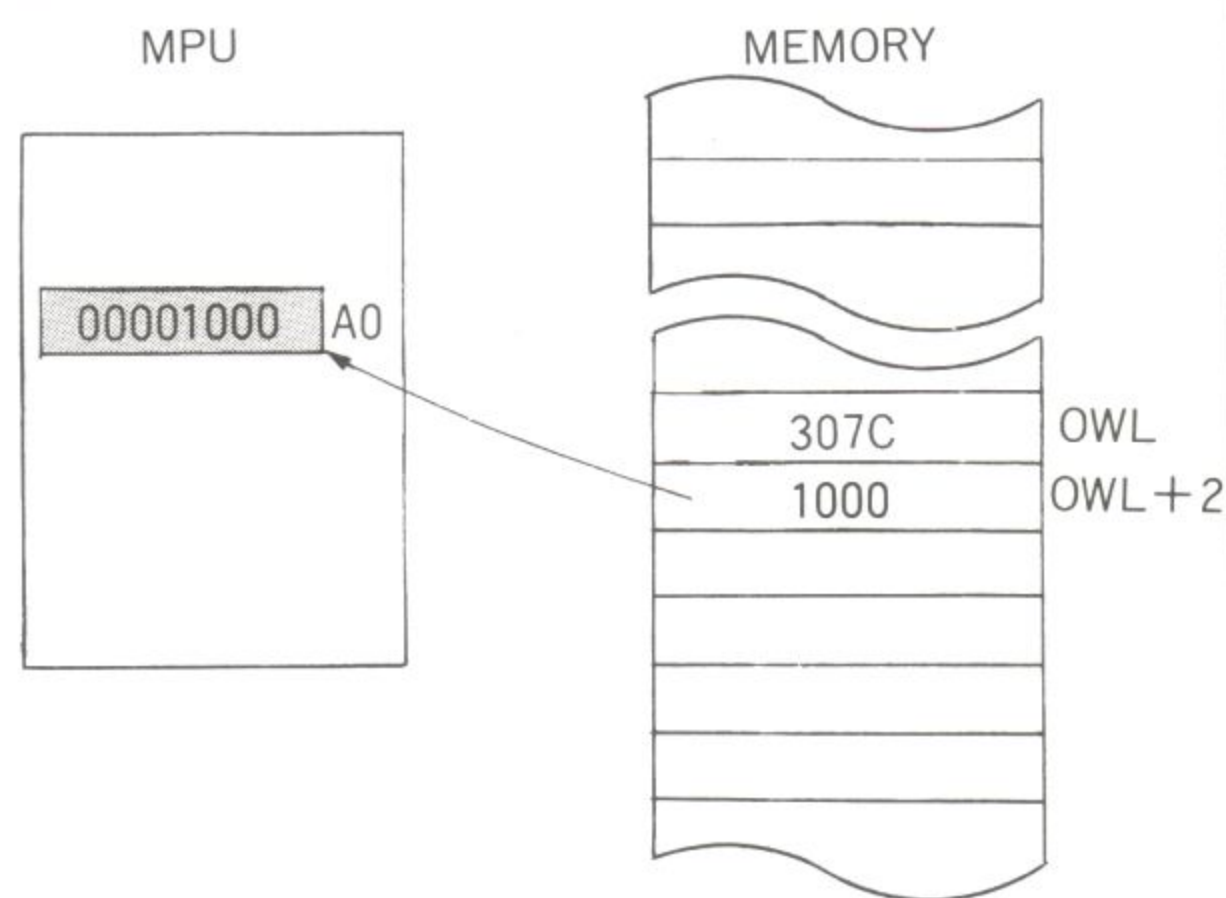
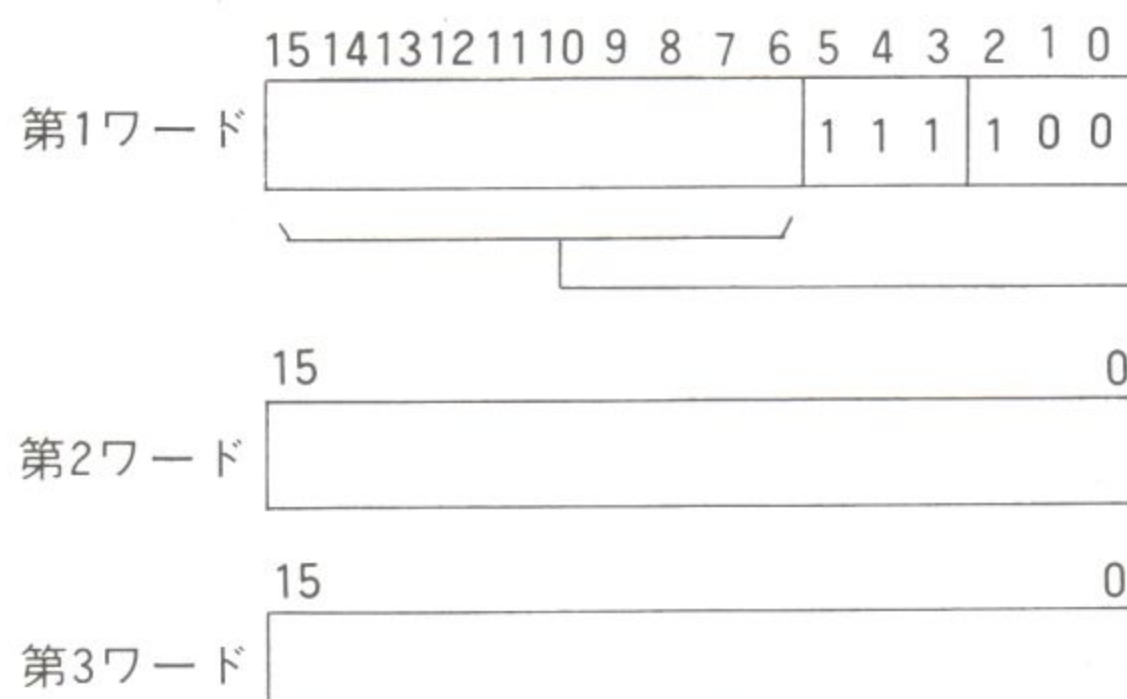
図1.41 イミディエイトデータ形式の例

機械語フォーマット

図1.42

note：第2，第3ワードはイミディエイト部であるが，データサイズがバイトまたはワードの場合は第3ワードは存在しない

13 SR／CCR形式

アセンブラ書式： **SRまたはCCR**

解説 対象となるデータは，ステータスレジスタ（SR）またはコンディションコードレジスタ（CCR，SRの下位バイト）の内容であるような形式ですが，SRを扱う命令の大部分は特権命令です．

この形式の命令セットを以下に示します（詳細は個別命令の説明を参照）．

ANDI	to	SR	ANDI	to	CCR
EORI	to	SR	EORI	to	CCR
ORI	to	SR	ORI	to	CCR
MOVE	to	SR	MOVE	to	CCR
MOVE	from	SR			

MOVE.W　$1020，SR

ソース側は絶対ショート形式，ディスティネーション側はSR形式によるもので，アドレス$1020から連続した2番地の内容が，SRへ転送される（本命令は特権命令で，スーパーバイザ状態でのみ使用可能）．

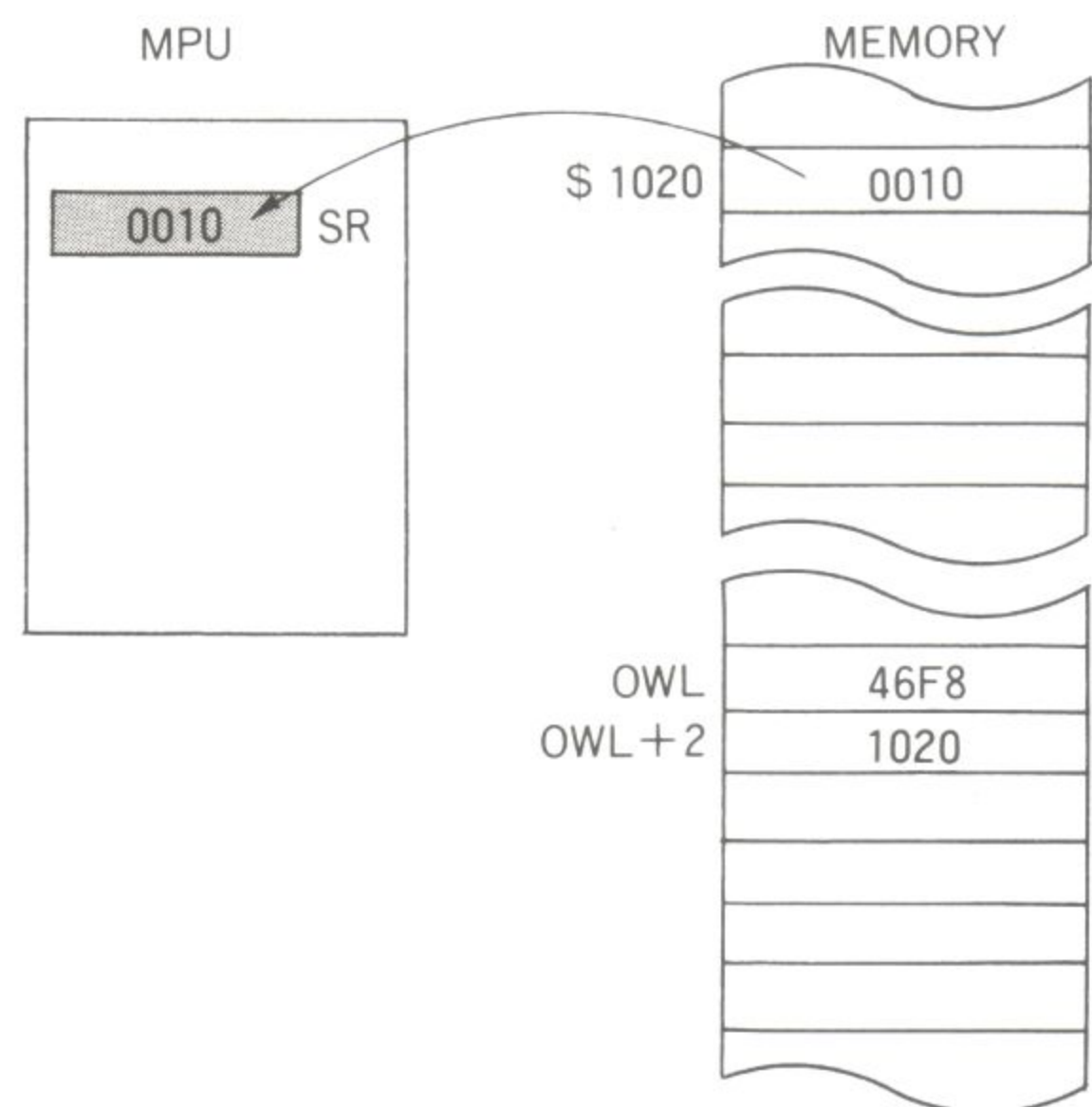
図1.43　SR/CCR形式の例

機械語フォーマット

図1.44

14 クイック・イミディエイト形式

解説　この形式はイミディエイト形式に分類されますが，イミディエイト値が拡張ワード内にセットされるのではなく，オペレーション・ワード内に存在するので，オブジェクトも短く，オペランドの取り込みが不要であるため，高速処理が期待できます．

サポートされている命令は，次の3通りです．

MOVEQ	：1バイトデータの高速転送
ADDQ	：1～8のインクリメント命令
SUBQ	：1～8のデクリメント命令

ところで68000にはINC（インクリメント），DEC（デクリメント）命令がありませんが，これを汎用化したものがADDQとSUBQであるわけで，1～8の値を1度に加減算できます．つまり，1命令で最高8回のインクリメントやデクリメントができるわけです．

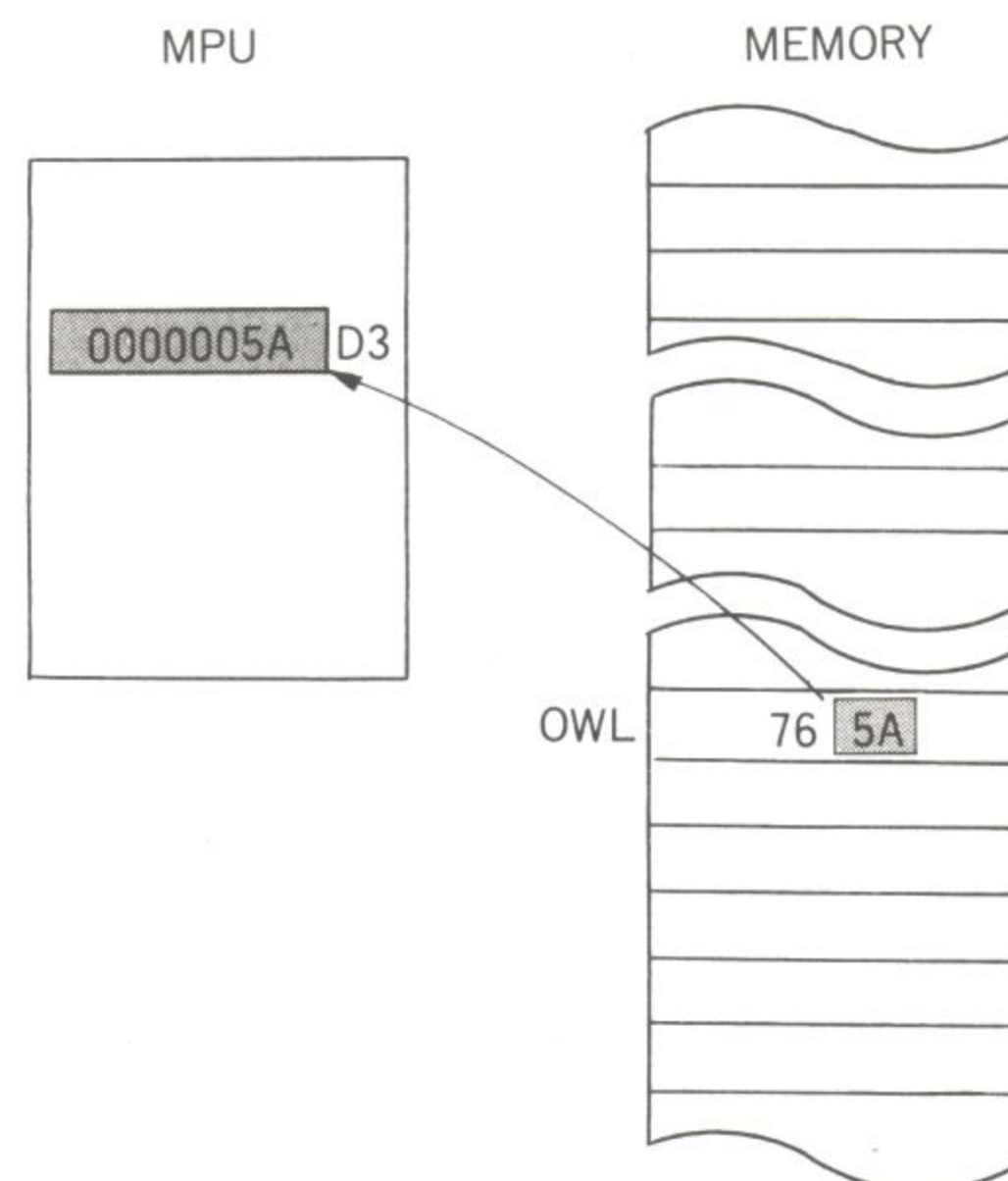
MOVE Q　#$5A，D3

ソース側はクイック・イミディエイト形式，ディスティネーション側はデータレジスタ直接形式によるもので，$5Aというイミディエイト値がD0へ転送される．

●MOVEQの特徴

① データは32ビットに符号拡張されてD3へ転送される．

② ディスティネーション（転送先）はデータレジスタに限定される．

図1.45　クイックイミディエイト形式の例

機械語フォーマット

※個別命令のセクション参照

15 インプリシット(Implicit, 暗黙的) 参照

解説 たとえば，RTS (リターン命令) はPC (プログラムカウンタ) とSP (スタックポインタ) を暗黙のうちに操作します．

このように，PC (プログラムカウンタ)，SP (システム・スタックポインタ)，SSP(スーパバイザ・スタックポインタ)，USP (ユーザスタックポインタ) SR(ステータスレジスタ)，などを暗黙的に参照する命令があり，**表1.11**は，この形式に分類される命令とその時に暗黙的に参照されるレジスタとを整理したものです．

表1.11　インプリシット (暗黙的) 命令参照

命令			暗黙的に参照されるレジスタ
Bcc		(Branch Conditional)	PC
BRA		(Branch Always)	PC
DBcc		(Test Condition, Decrement and Branch)	PC
JMP		(Jump)	PC
LINK		(Link and Allocate)	SP
PEA		(Push Effective Address)	SP
UNLK		(Unlink)	SP
MOVE	USP	(Move User Stack Pointer)	USP
MOVE	CCR	(Move Condition Code)	SR
MOVE	SR	(Move Status Register)	SR
BSR		(Branch to Subroutine)	PC, SP
JSR		(Jump to Subroutine)	PC, SP
RTS		(Return from Subroutine)	PC, SP
CHK		(Check Register against Bounds)	SSP, SR
DIVS		(Signed Divide)	SSP, SR
DIVU		(Unsigned Divide)	SSP, SR
TRAP		(Trap)	SSP, SR
TRAPV		(Trap on Overflow)	SSP, SR
RTE		(Return from Exception)	PC, SP, SR
RTR		(Return and Restore Condition Codes)	PC, SP, SR

PC　：プログラム・カウンタ
SP　：スタック・ポインタ
SR　：ステータス・レジスタ
SSP　：スーパーバイザ・スタック・ポインタ
USP　：ユーザ・スタック・ポインタ
CCR　：コンディション・コード・レジスタ

8 命令セットについて

マイクロプロセッサに要求される命令セットは，コンピュータがプログラムで動作する以上，数多く製品化されているプロセッサ間でも同様なはずです．この点で，すでにアドレッシングモードについては把握されたはずであり，68000の提供するプログラミング環境のすばらしさについて，理解されたことと思います．マイクロプロセッサの資料には命令の種類が列記されているわけですが，Ｚ－80や8086などに比較してかなり少ない命令になっています．重要なのは命令の数ではなく「１つ１つの命令がいかに強力であるか」ということであり，MOVE命令だけでもソースオペランドには12通り，ディスティネーションオペランドには９通り（MOVEAを含む）の指定が許され，108通りの転送方法が可能です．また，68000ではすべてのレジスタは汎用化され，特定のレジスタに対する命令を用意する必要もなかったわけで，必然的に命令数も少なくてすむわけです．

本セクションの意義

まず本セクションがなぜ必要か，つまり，「どう読んでいただくか」ですが，それには「本書がどのように展開されているか」ということを，今一度確認しなければなりません．

① 命令セットの詳細が収録されている〈第３部〉では個別命令を機能別に整理編集し，各命令の働きに関する詳細な解説が加えられ，オペレーションサイズ，許されるアドレッシングのすべて，機械語の構成の解析，オブジェクトサイズ，命令実行時のマシン・クロック・サイクル数など，必要とされる情報のすべてが網羅されている．

② 〈第２部プログラム編〉では，開発ツールとしてのアセンブラの文法に始まり，命令セットの詳細だけでは理解が困難であろうと判断される命令をピックアップし，具体的な例題を設定して解説し，報酬を前提としたプログラム開発を行う上で必要とされる基本的アルゴリズムや，汎用サブルーチンの作成をし，暗号のように思われる命令セットが有機的に機能する様子，つまり，プログラミングの解法について解説されている．

本セクションは，以上の２つの大きなセクションへの第一歩として，後に説明されるセクションを，より効果的に理解するために設けられています．

すべての命令を機能別に整理してみると，「ある共通した事実」を発見できます．そこには，汎用性を持った命令もあれば特定のオペランドを操作する命令もあるなど，「同じ命令群の横の関係」の知識も必要であると考え，それは，本セクションで解説する以外にないのです．そこで，データ転送命令なら，MOVE/MOVEA/MOVEQ　などの意味を，ちょっと離れたところからながめてみよう，というわけです．

各命令は，機能別に以下のような分類をしてあります．

① データ転送命令
② 整数算術演算命令
③ 論理演算命令
④ シフトおよびローテート命令
⑤ ビット演算命令
⑥ ２進化10進（BCD）命令
⑦ プログラム制御命令
⑧ システム制御命令
⑨ その他の命令

9 データ転送命令

データ転送命令はプログラマが最初に理解すべき命令であり，アドレッシングモードを理解する上でも重要です.

表1.12のような命令があります.

表1.12　データ転送命令

MOVE MOVEA MOVEQ MOVEM MOVEP	汎用のデータ転送を行う アドレスレジスタへの転送を行う 1バイトのイミディエイト値の転送を高速に行う MPUレジスタの退避／復帰を行う 周辺とのデータ転送を想定した命令
EXG	MPUレジスタ同士で内容を交換する
SWAP	データレジスタの上位と下位を交換する
LEA	実効アドレスの転送
PEA	実効アドレスをスタックへプッシュする
LINK UNLK	スタックフレームの確保を行う スタックフレームの解放を行う
CCR，SR，USPを操作するための転送命令	

以下，順を追って転送命令のポイントを整理してみます.

MOVE 〈ea〉，〈ea〉

汎用のデータ転送命令で，バイト，ワード，ロングワードのすべてのサイズと，すべてのアドレッシングがサポートされます．また，メモリ～メモリ転送がサポートされ，レジスタを介さず直接メモリからメモリへ転送できます.

以下のような特徴があります.

① 左側で指定した内容が右側へ転送され，極めて自然な記述である.

② ソース側は12通り，ディスティネーション側は8通りのアドレッシングがサポートされている.

③ 転送した内容がゼロならZフラグが，負ならNフラグがそれぞれセットされるが，Xフラグは保持されること，あるいはVとCフラグがクリアされるなど，転送命令とフラグ変化に無駄がなく，結果的にプログラムステップが少なくなる.

④ 転送先にアドレスレジスタを指定する命令はMOVEのサブセットであるMOVEAを使用する.

MOVEA 〈ea〉, An

MOVEAの〝A〞はAddressの〝A〞で，汎用のMOVE命令のうち転送先をアドレスレジスタに限定した命令であり，転送元は12通りの指定が許されます．

この命令はMOVEでサポートすればよさそうなものですが，アセンブラでは「アドレスという概念」は大変重要であり，アドレスを転送する命令が別になっているということは，「単なるデータ」と区別できる点で，バグを抑止できることのメリットの方が大であると思われます．

MOVEAは以下の点で汎用のMOVEと異なります．

① バイトサイズを扱わず，ワードまたはロングワードを扱う．
② ワードサイズではソース・オペランドの内容を符号拡張し，指定したアドレスレジスタへは32ビットのデータが格納される．
③ MOVE命令のようにフラグ変化しない．

MOVEQ #〈イミディエイト・データ〉, Dn

MOVEQの〝Q〞は Quick の〝Q〞で8ビットの即値をデータレジスタへ転送するものです．

特徴は次の通りです．

① 指定した8ビットのデータは32ビットに符号拡張され，データレジスタへ転送される．
② 指定した値がオブジェクトコード内に存在するので，オブジェクトコードが短く実行も高速である．

MOVEM 〈レジスタリスト〉, 〈ea〉
MOVEM 〈ea〉, 〈レジスタリスト〉

MOVEMの〝M〞はMultipleの〝M〞で，データレジスタ／アドレスレジスタのすべてを<ea>で指定したメモリエリアへ退避したりメモリエリアから復帰する命令ですが，レジスタリスト内にステータスレジスタを指定できません．

Z－80や8086では1つのレジスタを退避／復帰するには，その都度必要なだけPUSH/POPをしたものですが，68000ではこのように簡単に記述できるようになり，プログラマの負担が大きく軽減されるようになりました．

MOVEP Dn, d16 (An)
MOVEP d16 (An), Dn

MOVEPの〝P〞は Peripheral の〝P〞で，I／OとMPUデータレジスタとの転送を想定した命令です．

本命令を使用しなければ周辺とのデータ授受ができないわけではなく，汎用のMOVEの方が便利な場合もあるでしょうし，ハードウエア設計者の立場からは，さほど重要な命令ではないように思えます．これは，68000には通常のメモリ空間とI／O空間との区別はなく，ハードウエアの設計者がどのようにI／Oをサポートするかによって使うべき命令も異なるからです．

特徴は次の通りです.

① サイズはワードまたはロングワードであり，バイト転送はサポートされない.

② Anで指定したアドレスが偶数か奇数かによってイネーブルされるデータバスが決定され，偶数なら上位データバスが，奇数なら下位データバスがイネーブルされる.
偶数アドレスを指定しワード転送する場合，I／O側は偶数アドレスにインターフェースされているディバイスがセレクトされることになり，2n番地を指定したなら，2n，2n＋2番地がデータレジスタとの転送対象になる. 68000の習慣により，2n番地が上位バイト，2n＋2番地が下位バイトに対応する.
このように，2バイトの転送であれ4バイトの転送であれ，偶数アドレスと奇数アドレスが同時にセレクトされることはなく，いずれかの番地しか対象になりませんから，ワードまたはロングワードの転送がサポートされているものの，データバスはいずれか一方しか使用されず，16ビットの入／出力を同時に行うことはできない.

EXG 〈reg〉，〈reg〉

データレジスタ～データレジスタ，アドレスレジスタ～アドレスレジスタ，データレジスタ～アドレスレジスタ間で内容の交換をする命令で，サイズは32ビットです.

SWAP Dn

指定したデータレジスタの上位ワード（16ビット）と下位ワード（16ビット）を交換する命令です.

LEA 〈ea〉，An

＜ea＞で指定した実効アドレスを指定したアドレスレジスタへ転送しますが，アドレスは32ビットであり，Anの全32ビットが影響を受けます.

PEA 〈ea〉

LEAとスタックへのプッシュを1命令で行うもので，主にサブルーチンへアドレスを渡す時などに使用します.

＜ea＞で指定した実効アドレスをスタックへプッシュしますが，常にロングワードが操作されます.

LINK An，#〈エリア長〉
UNLK An

サブルーチンのセクションで具体例を説明しますが，68000の特徴的な命令であり，サブルーチンに必要とされるローカルエリアを確保したり解放するものです.

パラメータの受け渡しやローカルエリアの確保，リエントラントなサブルーチンを構築でき，Cコンパイラの設計者には極めて有用な命令ですが，アセンブラでプログラミングする際にも同様です（Z－80や8086で同様なことを実行しようとすれば，かなり面倒なことになります）.

MOVE 〈ea〉，CCR

ステータスレジスタの下位8ビットであるCCR（コンディション・コード・レジスタ）への転送を行うもので，ユーザプログラム内でフラグの復帰をしたり，直接フラグをセッ

ト／リセットできますから，サブルーチンからの復帰情報として，エラーが発生したらキャリ（C）を立ててもどるなど，5つのフラグ（コンディションコード）を同時に操作できます．

このようにフラグ操作も単なる転送命令に分類され，しかも<ea>で指定できるモードは11通りであり，柔軟なフラグ操作がサポートされています．

MOVE 〈ea〉, SR

これは特権命令であり，ステータスレジスタの操作を行うものです．

ビット15～8までの上位バイトには，トレース，スーパバイザ状態，割り込みマスクビットなどがマップされ，これらのビットを操作するために使用します．もちろん下位バイトであるCCRも操作できるのはいうまでもありません．

転送元は11通りの指定方式がサポートされています．

MOVE　SR, 〈ea〉

ステータスレジスタの内容を<ea>で指定した場所へ転送（退避）する命令ですが，本来ならMOVE　CCR，<ea>であるわけですが，これは68010でないと使用できません(手抜きと思われる)．それゆえ，68000ではSRを指定しても特権命令ではありません．

転送先は8通りのアドレッシングがサポートされています．

MOVE　USP, An
MOVE　An, USP

これらの命令は特権命令であり，スーパーバイザ状態でユーザ・スタック・ポインタの内容をアドレスレジスタへ転送したり，その内容を初期化したりする命令です．

システムスタック・ポインタは32ビット長のものが2セットあり，いずれもアドレスレジスタの7番目であるA7が割り当てられ，スーパーバイザ状態ではSSP（スーパーバイザ・スタック・ポインタ)，ユーザ状態ではUSP(ユーザ・スタック・ポインタ）と呼ばれます．

10 整数算術演算命令

算術演算命令の基本は，加算（ADD)，減算（SUB)，乗算（MUL)，除算（DIV)，比較（CMP)，であり，これらにはいくつかのサブセットが用意され，いずれの命令も強力なアドレシングがサポートされます．また，メモリをアキュムレータとしても使用できますから，メモリの内容と演算する場合でも，一度レジスタへ転送する必要がありません．

表1.13のような命令があります．

表1.13　算術演算命令

ADD	汎用の加算命令
ADDA	アドレスレジスタとの加算命令
ADDI	即値との加算命令
ADDQ	INC（インクリメント）を汎用化した命令
ADDX	倍精度の加算命令
SUB	汎用の減算命令
SUBA	アドレスレジスタの減算命令
SUBI	即値との減算命令
SUBQ	DEC（デクリメント）を汎用化した命令
SUBX	倍精度の減算命令
MULS	符号付き乗算命令
MULU	符号なし乗算命令
CMP	汎用の比較命令
CMPA	アドレスレジスタとの比較命令
CMPI	即値との比較命令
CMPM	連続した配列の比較命令
CLR	クリア命令
EXT	符号拡張命令
NEG	符号反転命令
NEGX	符号反転の拡張命令
TST	テスト命令（ゼロとの比較をする）
TAS	テスト・アンド・セット命令

■オペランドの記述について

オペランドの指定方法はインテル系（8086，Z－80）とオペランドの位置が逆で，カンマ（,）で分離されるオペランドの右側（ディスティネーション側）が主になり，結果も右側のオペランドへ格納されます．

この様子は，汎用命令だけでなく，類似のサブセット命令についても同様に解釈します．

① **加算命令の例**： ADD D０，(An)

D０の内容とAnでポイントされるメモリの内容が加算されるが，結果は右側のオペランドであるメモリへ格納され，D０の内容は変更されない．

② **減算命令の例**： SUB D０，D１

〝D１－D０〞が実行され，結果は右側のD１へ格納される．D０の内容は変更されない．

③ **比較命令の例**： CMP D０，D１

〝D１－D０〞が実行され，その結果がCCRへ反映される．ただし，フラグが演算結果に応じてセット／リセットされるだけで，D１へ格納されるわけではない．

ADD Dn，<ea>
ADD <ea>，Dn

汎用の加算命令で，ソース・オペランド(左側）とディスティネーション・オペランド(右側）の内容を加算し，結果をディスティネーション側へ格納します．

ADDA <ea>，An

ADDAの〝A〞はAddressの〝A〞でアドレスレジスタへの加算命令です．目的とするメモリアドレスの算出に使用されますが，このような目的ではフラグ変化しない方が便利であり，ADDA命令ではこのように配慮されています．

以下の点で汎用のADD命令と異なります．

① ディスティネーション・オペランドはアドレスレジスタAnであり，演算後，フラグ変化しない．

② オペレーションサイズは，ワード，ロングワードに限られ，バイトサイズを扱わない．

③ ワードサイズでは32ビットに符号拡張して演算される．

ADDI #<data>，<ea>

ADDIの〝I〞はImmediateの〝I〞でソース・オペランド(左側)のイミディエイト値 (即値) をディスティネーションへ加算し，結果をディスティネーションへ格納します．

ADDQ #<data>，<ea>

ADDQの〝Q〞はQuickの〝Q〞でインクリメント命令を汎用化した命令です．

指定可能なソース側イミディエイト値の範囲は１～８で，ディスティネーションに対して，最高８回のインクリメントを高速に一度で行うことができます．

ADDX Dn，Dn
ADDX －(An)，－(An)

ADDXの〝X〞はExtendの〝X〞で，倍精度加算命令です．

〝X〞はCCRのXビットをソースとともに加算することを意味し，特に〝－ (An)〞のアドレッシングは，メモリ配列の加算に便利な命令です．

SUB　Dn, <ea>
SUB　<ea>, Dn

汎用の減算命令であり，(ディスティネーションオペランド)－（ソースオペランド）を実行し，結果をディスティネーションへ格納します．

SUBA　<ea>, An

SUBAの〝A〟はAddressの〝A〟でアドレスレジスタの内容を減じる命令です．
汎用のSUB命令とは以下の点で異なります．

① オペレーションサイズはバイトサイズを扱わず，ワードまたはロングワードに限定される．
② ワードサイズではソースオペランドを32ビットに符号拡張して演算される．
③ SUBのようにフラグ変化しない．

SUBI　#<data>, <ea>

SUBIの〝I〟はImmediateの〝I〟でディスティネーションからイミディエイト値（即値）を減算し，結果をディスティネーションへ格納します．

SUBQ　#<data>, <ea>

SUBQの〝Q〟はQuickの〝Q〟で，デクリメント命令を汎用化した命令です．指定可能なソース側イミディエイト値の範囲は１～８で，ディスティネーションに対して，最高８回のデクリメントを高速に一度で行うことができます．

SUBX　Dn, Dn
SUBX　－(An), －(An)

SUBXの〝X〟はExtendの〝X〟で，倍精度減算命令です．
〝X〟はディスティネーションからソースとXビットを減じることを意味します．
特に〝－（An)〟のアドレッシングは，メモリ配列の減算に便利な命令です．

MULS　<ea>, Dn
MULU　<ea>, Dn

〝S〟はサイン（符号付き)，〝U〟はアンサイン(符号なし)を意味し，符号付きおよび符号なしの乗算命令です．要点は次の通りです．

① ソース，ディスティネーションともに下位ワードが演算対象になる．
② 結果はディスティネーションのDnへ32ビットで格納される．

DIVS　<ea>, Dn
DIVU　<ea>, Dn

〝S〟はサイン（符号付き)，〝U〟はアンサイン（符号なし）を意味し，符号付きおよび符合なしの除算命令です．要点は次の通りです．
①（ディスティネーション：32ビット)÷(ソース：下位16ビット）を行う．
② ディスティネーションのDnの上位ワードへ余りを，下位ワードへ商を格納する．

③ ゼロで割るとTRAPが発生し，命令完了前にオーバーフローを検出するとVビットがセットされ，除算は実行されない（オペランドは影響を受けない）.

CMP <ea>，Dn

汎用の比較命令であり，(ディスティネーション)－（ソース）を実行し，両者の比較結果はCCRへ反映されますが，減算命令と異なり，演算結果はディスティネーションのDnへ格納されません.

CMPA <ea>，An

CMPAの〝A″はAddressの〝A″で，ディスティネーションで指定したアドレスレジスタとの比較命令で，以下の点で汎用のCMP命令と異なります.（比較命令であるから，もちろんフラグ変化する.）

① オペレーションサイズにはバイトを指定できず，ワードおよびロングワードのいずれかを指定する.

② サイズがワードであれば，ソースオペランドは32ビットに符号拡張され，ディスティネーション（アドレスレジスタ）も32ビットを使用し，比較を行う.

CMPI #<data>，<ea>

CMPIの〝I″はImmediateの〝I″で，ディスティネーションオペランドと即値との比較命令です.

CMPM (An)+，(An)+

CMPMの〝M″はMemoryの〝M″でメモリ配列（メモリブロック）専用の比較あるいは検索命令です.

CLR <ea>

クリア命令であり，Z－80や8086では〝XOR″などで代用していました.

フラグ変化ですが，本命令はゼロをオペランドに書き込むので，Zフラグが立ち，N，V，Cはリセットされます.

EXT Dn

Dnの内容を符号拡張します.

すなわち，オペランド・サイズにバイトを指定すれば，ワードに，ワードを指定すれば，ロングワードに，それぞれ符号拡張します.

指定できるオペランドはデータレジスタのみで，メモリオペランドを指定できません.

NEG <ea>
NEGX <ea>

データの符号反転を行う命令で，ゼロからオペランドを減じます．NEGXでは，さらにXビットも減じられます．たとえば，<ea>で指定される内容が十進で〝10″なら，NEGにより〝－10″になります.

TST <ea>

比較命令の限定版で，オペランドは常にゼロと比較され，結果として得られるステータスは，Nビット（正／負）とZビット（ゼロか否か）です．ただし，VおよびCビットはビットはクリアされます.

TAS <ea>

マルチプロセッサ間の同期に使用されますが，メモリサイクルは，ハード的に，リード～モディファイ～ライト，サイクルがサポートされねばなりません.

11 論理演算命令

論理演算命令（AND, OR, EOR, NOT）は，すべての整数データオペランドに対して有効であり，イミディエイト値を扱うサブセットとして，ANDI, ORI, EORI, などがあり，このサブセットのディスティネーション・オペランドがSR（ステータスレジスタ）の場合は，特権命令になります．

オペランドの記述については算術演算命令と同じであり，結果は第2オペランドであるディスティネーションへ格納され，ソース・オペランドの値は変更されません．

また，いずれのオペランドにもアドレスレジスタを指定できず，アドレスレジスタは論理演算の対象から除外されます．

表1.14のような命令があります．

表1.14　論理演算命令

AND	汎用のAND命令
ANDI	即値とのAND命令
ANDI to CCR	即値とCCRの内容とのANDをとり，CCRを操作するための命令
ANDI to SR	即値とSRの内容とのANDをとり，SRを操作するための命令で，これは特権命令である
OR	汎用のOR命令
ORI	即値とのOR命令
ORI to CCR	即値とCCRの内容とのORをとり，CCRを操作するための命令
ORI to SR	即値とCRの内容とのORをとり，SRを操作するための命令で，これは特権命令である
EOR	汎用のEOR命令
EORI	即値とのEOR命令
EORI to CCR	即値とCCRの内容とのEORをとり，CCRを操作するための命令
EORI to SR	即値とCCRの内容とのEORをとり，SRを操作するための命令で，これは特権命令である
NOT	1の補数（すべてのビットを反転）をとる命令

AND　Dn, <ea>
AND　<ea>, Dn

汎用のAND命令であり，ディスティネーション・オペランドとソース・オペランドとの論理積をとり，結果をディスティネーションへ格納しますが，アドレスレジスタの内容をオペランド(ソース/ディスティネーションとも)に指定できません．

ANDI #<data>, <ea>
ANDI #<data>, CCR
ANDI #<data>, SR

いずれもイミディエイト値とディスティネーションとの論理積をとり，結果をディスティネーションへ格納します．

① ディスティネーションが<ea>であれば，データ・可変モードのアドレッシングが許され，8通りのアクセス方法が可能であるが，アドレスレジスタの内容をオペランド（ソース／ディスティネーションとも）に指定できない．
② ディスティネーションがCCRであれば，ユーザプログラム内でCCRの各ビットを操作できる．
③ ディスティネーションがSRであればスーパーバイザ状態でSRの各ビットを操作できる．

OR Dn, <ea>
OR <ea>, Dn

汎用のOR命令であり，ディスティネーションオペランドとソースオペランドとの論理和をとり，結果をディスティネーションへ格納しますが，アドレスレジスタの内容をオペランド（ソース／ディスティネーションとも）に指定できません．

ORI #<data>, <ea>
ORI #<data>, CCR
ORI #<data>, SR

いずれもイミディエイト値とディスティネーションとの論理和をとり，結果をディスティネーションへ格納しますが，アドレスレジスタの内容をオペランド（ソース／ディスティネーションとも）に指定できません．

① ディスティネーションが<ea>であれば，データ・可変モードのアドレッシングが許され，8通りのアクセス方法が可能である．
② ディスティネーションがCCRであれば，ユーザプログラム内でCCRの各ビットを操作できる．
③ ディスティネーションがSRであればスーパーバイザ状態でSRの各ビットを操作できる．

EOR Dn, <ea>

汎用のEOR命令であり，ディスティネーション・オペランドとソース・オペランドとの排他的論理和をとり，結果をディスティネーションへ格納しますが，アドレスレジスタの内容をオペランド（ソース／ディスティネーションとも）に指定できません．

また，EOR　<ea>，Dnという表現は許されません．

EORI　#<data>，<ea>
EORI　#<data>，CCR
EORI　#<data>，SR

いずれもイミディエイト値とディスティネーションとの排他的論理和をとり，結果をディスティネーションへ格納します．

① ディスティネーションが<ea>であれば，データ・可変モードのアドレッシングが許され，8通りのアクセス方法が可能であるが，アドレスレジスタの内容をオペランド（ソース／ディスティネーションとも）に指定できない．
② ディスティネーションがCCRであれば，ユーザプログラム内で，CCRの各ビットを操作できる．
③ ディスティネーションがSRであればスーパーバイザ状態でSRの各ビットを操作できる．

NOT　<ea>

ディスティネーション・オペランドに対して1の補数(全ビットを反転）をとり，結果をディスティネーションへ格納しますが，アドレスレジスタをオペランドに指定できません．

12 シフトおよびローテート命令

シフト／ローテートとは，「ビット単位の移動や回転をするものです」と説明されても，「それは何の役に立つのですか」と質問されるかもしれません．具体的な応用例は<プログラミング編>で取り上げるものとし，とにかく，ビット単位でシフトしたり，ローテート（回転）する命令があることを知っておいてください．

表1.15のような命令があります．

表1.15 シフトおよびローテート命令

命令	内容
ASL ASR	左へ算術シフト 右へ算術シフト
LSL LSR	左へ論理シフト 右へ論理シフト
ROL ROR	左へのローテート 右へのローテート
ROXL ROXR	Xビットを含めた左へのローテート Xビットを含めた右へのローテート

いずれの命令も指定できるオペランドのバリエーションは3通りで，これらの役割について整理しておきます．

オペランドが　Dn，Dn　である場合 [例：ASL　Dn，Dn]

① ソース側のデータレジスタはシフトまたはローテート時のカウンタであり，一度にシフトまたはローテートするビット数（0～63）を指定する．
② ディスティネーション側には操作対象のデータレジスタ番号を指定する．
③ オペレーションサイズは，バイト／ワード／ロングワードである．

オペランドが　#<data>，Dn　である場合 [例：ASL　#<data>，Dn]

① ソース側のイミディエイト値で一度にシフトまたはローテートするビット数（1～8）を指定する．
② ディスティネーション側には操作対象のデータレジスタ番号を指定する．
③ オペレーションサイズは，バイト／ワード／ロングワードである．

オペランドが <ea> である場合 [例：ASL <ea>]

① 一度に指定できるシフトまたはローテートのビット数は１（イチ）である．
② オペランドはメモリ・可変アドレッシングモードが適用され，７通りのアクセス方式が可能である．
③ オペランドサイズはワードに限定される．

表1.16 シフトおよびローテートのニーモニック構成

ASd	AS は Arithmetic Shift（算術シフト）を意味する． d は シフト方向を意味するＲ(右)またはＬ(左)である．
LSd	LS は Logical Shift（論理シフト）を意味する． d は シフト方向を意味するＲ(右)またはＬ(左)である．
ROd	RO は Rotate（ローテート）を意味する． d は ローテート方向を意味するＲ(右)またはＬ(左)である．
ROXd	RO は Rotate（ローテート）を意味する． X は Ｘビットも含めてローテートすることを意味する． d は ローテート方向を意味するＲ(右)またはＬ(左)である．

ASd Dn，Dn [dはＬ／Ｒを意味する]
ASd #<data>，Dn
ASd <ea>

算術シフト命令であり，ディスティネーション・オペランドのビット群を指定方向へ算術シフトし，オペランドの外へ押し出されたビットは，ＸビットとＣビットへ反映されます．

LSd Dn，Dn [dはＬ／Ｒを意味する]
LSd #<data>，Dn
LSd <ea>

論理シフト命令であり，ディスティネーション・オペランドのビット群を指定方向へ論理シフトし，オペランドの外へ押し出されたビットは，ＸビットとＣビットへ反映されます．

ROd Dn，Dn [dはＬ／Ｒを意味する]
ROd #<data>，Dn
ROd <ea>

ディスティネーション・オペランドのビット群を指定方向へローテート(回転)しますが，押し出されたビットは，Ｃビットへ反映されます．

ROXd Dn，Dn [dはＬ／Ｒを意味する]
ROXd #<data>，Dn
ROXd <ea>

ディスティネーション・オペランドのビット群を指定方向へローテート（回転）しますが，Ｘビットも含めてローテートされ，押し出されたビットは，ＸビットとＣビットへ反映されます．

13 ビット演算命令

ビット演算命令には，BTST（ビットテスト），BSET（ビットセット），BCLR（ビットクリア），BCHG（ビットチェンジ），の4つの基本命令があり，ビットごとの演算を行うことができます．

表1.17のような命令があります．

表1.17　ビット演算命令

BTST	指定ビットがゼロ（0）であるか否かをテストする．
BSET	指定ビットをセット（1）する．
BCLR	指定ビットをクリア（0）する．
BCHG	指定ビットを反転（チェンジ）する．

各命令の仕様を検討してみますと，68000のアーキテクチャにしては，やや不合理な面をもっていますので，まずこの点について整理しておくことにします．

ビット演算命令は，本来なら，次の3つの項目を指定できるはずですが，オペランドのサイズはハードウェアで決定され，指定できません．

① オペランドのサイズ
② 演算の対象となるビット番号（ソース側で指定する）
③ 演算の対象となる内容（ディスティネーション側で指定する）

結論を先に言ってしまうと，演算対象がデータレジスタであれば，サイズはロングワードとみなされ，ビット31～0が命令の対象となります．一方，メモリの内容が演算対象である場合，サイズはバイトとみなされ，ビット7～0が演算対象となります（ワードというサイズは存在しません）．

BTSTを例にとってこの問題にふれますが，その他のビット演算命令も操作内容が異なるだけで，基本的な考え方は同様です．オペランドのバリエーションは，以下の2つです．

```
BTST     Dn，Dn          ………＜1＞
BTST     #<data>,<ea>    ………＜2＞
BTS.L    #$31，(A0)      ………＜2'＞
```

＜1＞では，ディスティネーションがデータレジスタであることから，サイズはロングワードであり，テストしたいビット番号を指定するために，ソース側のデータレジスタには，31～0の値をセットすることができます．

＜2＞では，ディスティネーション側で操作される内容が何者かを指定するわけですが，＜1＞と同様にデータレジスタを指定する場合と，メモリオペランドを指定する場合の2通りがあり，サイズに関しては，データレジスタならロングワード，メモリならバイトとみなされ，これに応じて，#<data>で指定する値も決定されてしまいます．

＜2'＞は，A 0でポイントされるメモリの内容のビット31をテストするものですが，メモリオペランドはバイトであり，ビット31は存在しないことになっているので，このようなテストをするには，一度データレジスタへ読み込んでから，＜1＞の方式を適用しなければなりません．

これらを以下に整理します。

ビット演算の対象	指定できるビットの範囲	オペレーションサイズ
データレジスタ	ビット31〜0	ロングワードに固定される
メモリ	ビット7〜0	バイトに固定される

BTST Dn，<ea>
BTST #<data>，<ea>

ディスティネーション・オペランドの指定ビットがゼロ（0）であるか否かをテストし，結果をZビットへ反映します．テストしたいビット番号は，ソースオペランドで指定します．

BSET Dn，<ea>
BSET #<data>，<ea>

ディスティネーション・オペランドの指定ビットがゼロ（0）であるか否かをテストし，結果をZビットへ反映します．その後，指定ビットをセット（1）します．セットしたいビット番号は，ソースオペランドで指定します（単に指定ビットをセットするのではなく，セットする直前のビット状態をZビットに求めています）．

BCLR Dn，<ea>
BCLR #<data>，<ea>

ディスティネーション・オペランドの指定ビットがゼロ（0）であるか否かをテストし，結果をZビットへ反映します．その後，指定ビットをクリア（0）します．クリアしたいビット番号は，ソースオペランドで指定します（単に指定ビットをクリアするのではなく，クリアする直前のビット状態をZビットに求めています）．

BCHG Dn，<ea>
BCHG #<data>，<ea>

ディスティネーション・オペランドの指定ビットがゼロ（0）であるか否かをテストし，結果をZビットへ反映します．その後，指定ビットを反転します．反転したいビット番号は，ソースオペランドで指定します（単に指定ビットを反転するのではなく，反転する直前のビット状態をZビットに求めています）．

14 2進化10進命令

2進化10進（Binary Coded Decimal）命令には，ABCD（加算命令），SBCD（減算命令），NBCD（符号反転命令）があります．

表1.18のような命令です．

表1.18　2進化10進命令

ABCD	BCD加算
SBCD	BCD減算
NBCD	BCD値の1の補数をとる（符号反転）

各命令に共通した項目を次に整理します．

① オペレーションサイズはバイトである．
② 多数桁の2進化10進演算をサポートするため，Xビット（拡張フラグ）を含めた演算を行う．

NBCD以外の加算（ABCD）減算（SBCD）命令のオペランドは，次の2通りの指定が許されます．

① データレジスタ～データレジスタ間のBCD加算または減算．
② プリデクリメント形式によるメモリ～メモリ間のBCD加算または減算．

ABCD Dn，Dn
ABCD －(An)，－(An)

ディスティネーション・オペランドに，ソース・オペランドとXビットを加算し，結果をディスティネーションへ格納する命令で，加算は2進化10進数の演算によって行われます．

SBCD Dn，Dn
SBCD －(An)，－(An)

ディスティネーション・オペランドから，ソース・オペランドとXビットを減算し，結果をディスティネーションへ格納する命令で，減算は2進化10進数の演算によって行われます．

NBCD <ea>

ゼロ（0）から，ディスティネーション・オペランドとXビットを減算し，結果をディスティネーションへ格納する命令で，減算は2進化10進数の演算によって行われます．

15 プログラム制御命令

プログラム制御命令は大変重要なセクションでもあり，詳細に関しては，それなりのプログラミング環境を設定した上でその働きを納得できるような解説をしますので，ここでは，プログラム制御命令のポイントを簡潔に整理しておくことにします．

そこで，条件付き分岐命令なら，どのような条件を指定できるのか，ブランチとジャンプとの相違点，単なる分岐とサブルーチン分岐との相違点など，命令の特徴（ポイント）を把握しておけば十分でしょう．

通常のプログラムは，PC（プログラムカウンタ）で指定されたロケーションから順に実行され，この間，記述されている命令はスキップされません．一方，プログラム制御命令は，たとえば，ある値を比較し，その結果に応じた処理ルーチンへ分岐するとか，通常の処理の流れを，強制的に別の方向へスイッチするような命令を意味します．逆に，このような命令がなければ，コンピュータは（我々にとって）まったくつまらないサービスしかできないことになります．

表1.19のような命令があります．

表1.19　プログラム制御命令の分類

分類［1］	条件を指定し，その条件に応じて分岐する命令
Bcc	条件付き分岐命令（分岐範囲のオフセットは2バイト以内）
DBcc	ループ制御命令（分岐範囲のオフセットは2バイト以内）
Scc	条件セット命令（本命令は分岐命令ではない）
分類［2］	**分岐するのに，条件を必要としない命令**
BRA	無条件分岐命令（分岐範囲のオフセットは2バイト以内）
BSR	サブルーチン分岐命令（分岐範囲のオフセットは2バイト以内）
JMP	無条件分岐命令（16Mバイト全域）
JSR	サブルーチン分岐命令（16Mバイト全域）
分類［3］	**リターン命令**
RTS	サブルーチンからの復帰命令（CCRを復帰しない）
RTR	サブルーチンからの復帰命令（CCRも復帰する）

表1.20　ニーモニックのキーワード

B	Branchのイニシャルで，分岐範囲に制限（オフセットは2バイト）がある命令．
J	Jumpのイニシャルで，全域に分岐できる命令．
SR	Sub－RoutineのSRでサブルーチンコール命令．
R	Returnのイニシャルでサブルーチンからの復帰命令．

Bcc <label>

典型的な条件付き分岐命令で，cc (Condition Code) で指定された条件をテストし，満足されていればラベルで指定されたロケーションへ分岐し，それ以外は，本命令の次の命令を実行します．

コンディションコードを変化させる命令とペアで使用し，たとえば，ある値を比較し，その結果に従って分岐させたい場合などに使用します．この時のccには14通りの条件を指定でき，等しいときに所定の場所（LABEL）へ分岐させるには，

BEQ　LABEL

と記述します．

表21　Bcc命令で指定できる条件

条　件	意　味	
CC CS	キャリ・クリア キャリ・セット	(Cary Clear) (Carry Set)
EQ NE	等しい 等しくない	(Equal) (Not Equal)
GE LE	大きいかまたは等しい 小さいかまたは等しい	(Greater or Equal) (Less or Equal)
GT LT	より大きい より小さい	(Greater Than) (Less Than)
HI LS	ハイ ローあるいは同じ	(High) (Low or Same)
MI PL	マイナス プラス	(Minus) (Plus)
VC VS	オーバーフローなし オーバーフローあり	(OVerflow Clear) (OVerflow Set)

DBcc Dn, <label>

ループ制御命令であり，ある区間の命令を繰り返し実行させたい場合に使用します．条件付き分岐命令と2～3の他の命令を組み合わせることによって，本命令と同様な動作を実現できますが，68000には，このように強力なループ制御命令が用意されています．

① ループの終了条件（ccのことを意味）をテストし，満足されていれば分岐せず，次の命令を実行する．つまり，ループを抜け出すので，本命令の役割は，この段階で終了する．終了条件が満たされていなければ，「満たされるまで何かをさせる」というのが，「プログラマの願い」であり，②へ行く．

② 2番目の終了条件であるソースオペランドで指定したデータレジスタの内容をチェックする．ここでは，データレジスタをデクリメントし，マイナス1（－1）ならループを抜け出し，本命令の次の命令を実行する．したがって，本命令の役割は，この段階でも終了する．そうでない場合，指定したラベルへ分岐（ループを意味）する．

以上のように，繰り返し回数と**表1.22**コンディション・コード（16通り）の２つの条件を指定でき，非常に柔軟な制御構造を容易に実現できます．

表1.22　DBcc命令で指定できる条件

条　件	意　味	
CC	キャリ・クリア	(Cary Clear)
CS	キャリ・セット	(Carry Set)
EQ	等しい	(Equal)
NE	等しくない	(Not Equal)
GE	大きいかまたは等しい	(Greater or Equal)
LE	小さいかまたは等しい	(Less or Equal)
GT	より大きい	(Greater Than)
LT	より小さい	(Less Than)
HI	ハイ	(High)
LS	ローあるいは同じ	(Low or Same)
MI	マイナス	(Minus)
PL	プラス	(Plus)
VC	オーバーフローなし	(overflow Clear)
VS	オーバーフローあり	(overflow Set)
T	常に真	(Always True)
F*	常に偽	(Always False)

＊　DBF　ではなく，DBRAというニーモニックが許される．

Scc　<ea>

指定した条件によって，オペランドのすべてのビットをセット／リセットする命令で，プログラム自身に必要なフラグ操作を支援するものです．

まず指定されたコンディション・コード（cc）をテストし，条件が満たされれば，<ea>で指定したオペランドの全ビット（８ビット）をセットし，そうでなければリセットします．Scc命令で指定できる条件は，DBcc（**表1.22**）と同じです．

BRA　<label>

無条件分岐命令で，指定されたラベルへ分岐します．

BSR　<label>

サブルーチン分岐命令で，本命令直後のアドレスをシステムスタックへプッシュし，指定ラベルへ分岐します．

JMP　<ea>

<ea>で指定されたロケーションへ分岐します．

注） A０に＄5000というアドレスデータが格納されるとき，
JMP（A０）
により，プログラムの制御は＄5000番地へ移行する．

JSR <ea>

サブルーチン分岐命令で，本命令直後のアドレスをシステムスタックへプッシュし，<ea>で指定されたサブルーチンコール先へ分岐します．

注）A 0に＄5000というアドレスデータが格納されているとき，

JSR（A 0）

により，プログラムの制御は＄5000番地へ移行する．

RTS

サブルーチンからの復帰命令で，BSRまたはJSRで呼び出されるサブルーチンは，本命令でメイン・ルーチンへもどります．

RTR

RTSと同様な命令ですが，リターン時にCCRも操作される点が異なります．

16 システム制御命令

システム制御命令は，名称の通り，システムの制御に関する命令で，このような知識は，「体系的なプログラム」を作成する場合に要求されるものです．

68000を搭載したMPUボード上に搭載するプログラムを，電源が投入された時点から記述するようなレベルのソフトウエア，たとえば，システム・モニタとか，比較的小規模な機器組み込み用のサービスプログラム，あるいはOS（オペレーティング·システム）などの開発には，どうしても，このような命令が介在することになります．

システム制御に分類される命令群は，**表1.23**の3つに大別できます．

表1.23　システム制御命令の分類

命令	内容
1：CCR（フラグ）操作命令	
ANDI to CCR	CCRと即値との論理積をCCRへ格納
ORI to CCR	CCRと即値との論理和をCCRへ格納
EORI to CCR	CCRと即値との排他的論理和をCCRへ格納
MOVE to CCR	CCRへの転送
MOVE from SR	CCRの参照（システムバイトも参照可）
2：トラップ発生命令	
TRAP	無条件にトラップを発生
TRAPV	CCRのVビット（オーバーフロー）が有効である場合にトラップを発生
CHK	メモリの境界が犯された場合にトラップを発生
3：特権命令	
ANDI to SR	SRと即値との論理積をSRへ格納
ORI to SR	SRと即値との論理和をSRへ格納
EORI to SR	SRと即値との排他的論理和をSRへ格納
MOVE to USP	USPの設定
MOVE from USP	USPの参照
RESET	外部ディバイスのリセット
RTE	例外処理からの復帰
STOP	プロセッサ停止

以下，これらの概要について整理します．

［1］　CCR（フラグ）操作命令

CCRを操作する命令に関しては，すでに転送命令や論理演算の概要でもふれましたが，SRを参照するだけで変更しない命令に関しては，68000では特権化されていないので，実質的にCCRの操作に分類しました．

MOVE from SRの用途としては，サブルーチンの前処理として，フラグをスタックへプッシュすることです．その他の命令群の用途は，いうまでもなく，CCRの各ビットを操作するもので，具体的には，

① 必要なビットを保存し，特定ビットをクリア／セット／反転する
② 全ビットをクリア／セット／反転する
ということです．

［2］ トラップ命令

トラップ命令は，ユーザ状態からスーパーバイザ状態へ移行させ，トラップに対するサービスを要求する場合に使用され，用途としては以下の2つが考えられます．
① OSのファンクションコールを行う場合
② オーバーフローやゼロ除算などの処理ルーチンを呼び出す場合

TRAP #<ベクタ番号>

OSのファンクションコールが典型的な用途で，無条件にトラップを発生させます．ベクタ番号によって，どのようなサービスを要求するのかを指定し，0番～15番までの16通りを指定でき，これらの番号は例外ベクタ番号の32番～47番に対応します．
例外ベクタ領域には，トラップが発生した場合に，どこへプログラムの制御を移行するのかという情報，つまり，スーパーバイザ状態で行うべきサービスルーチンの処理アドレスが格納されていなければなりません．

TRAPV

演算のオーバーフローをチェックする命令で，Vビットが有効（セット）であるときに本命令を実行すると，トラップを発生します．例外ベクタには7番が割り当てられ，$1Cから格納されているアドレスを取り出し，そこへ制御を移行します．
Vビットがクリアされオーバーフローが発生していなければ，トラップは発生せず，単に次の命令へ制御が移ります．

CHK <ea>，Dn

CHKはメモリアクセスの境界チェックを行うもので，Dnとゼロ（0）との比較，Dnと< ea>で指定した内容の比較をします．これらの比較結果に従ってトラップが発生します．また，どのようなケースでトラップが発生したのかをチェックできるように，Nビットも操作されます．つまり，

Dn ＜ 0　　　　　　　なら，Nビットをセットしてトラップを発生
Dn ＞（<ea>）　　　なら，Nビットをクリアしてトラップを発生
0 ≦ Dn ≦（<ea>）なら，トラップは発生しない（Nビットは不定）

トラップが発生すると，例外ベクタ番号には6番が割り当てられているので，＄18から格納されている処理アドレスを取り出し，そこへ制御を移し，トラップが発生しなければ，単に次の命令へ制御が移ります．

［3］ 特権命令

私たちは，OS(オペレーティング・システム)を意識することなく，コンピュータに様々な仕事をさせています．

事実，ワードプロセッサのマニュアルに従った操作をするだけで，文章の作成や印字や保存をしているわけですが，コンピュータのソフトウエアには，OS と呼ばれるシステムレベルのプログラムと，ワードプロセッサに代表されるアプリケーション・プログラムの2つがあり，特権命令は，OS の信頼性を一層向上させるために用意されているものと，考えることができます．

それでは，「OS とは何か」ということですが，コンピュータを使う上で，効果的にコンピュータを運用するために必要な仕事をさせるプログラム，つまり，我々がコンピュータを（できればコンピュータということを意識せずに）使う上では不可欠な，コンピュータの基本運用プログラムであるといえます．

抽象的な表現になってしまいましたが，OS に興味を持たれ，そして OS の開発に従事されれば，スーパーバイザとか特権命令は，なくてはならないものであることを理解できるはずです．

68000には特権命令に分類されるいくつかの命令群があり，これらは，スーパーバイザ状態でのみ実行可能であり，ユーザ状態での実行から保護されています．そして，スーパーバイザ状態とユーザ状態とは，以下のように SR の S ビットで管理されます．

すなわち，

① Sビットが有効（セット）　——　スーパーバイザ状態
② Sビットが無効（リセット）——　ユーザ状態

特権命令は，すでに転送命令や論理演算命令の特殊な部分として「特権命令」と説明してきたものを含め，以下のような3通りに分類することができますが，これらの命令群をユーザ状態で実行すると，特権違反として例外処理へ移行（トラップの発生）します．

① SR を操作する命令群
② USP を操作する命令群
③ タスクの推移に関する命令群

【SR を操作する命令群】

SR を操作する命令群の用途は，次のようなものです．

① プログラム実行状態をスーパーバイザに変更する
② トレース処理
③ 割り込み処理
④ CCR の操作

【USP を操作する命令群】

スーパーバイザ状態で USP をアクセスする命令で，USP の設定や参照は以下のように行います．

```
MOVE.L  An,USP   ：スーパーバイザ状態での USP の設定
MOVE.L  USP,An   ：スーパーバイザ状態での USP の参照
```

スーパーバイザ状態でユーザスタックを操作するには，″USP″ という特別なニーモニックを指定しなければなりません．また，アドレスレジスタにＡ７と記述すれば，スーパーバイザ・スタックポインタが対象になるので，アドレスレジスタの番号ｎは０～６に意味

があります．このようにする理由は，システムスタックポインタにはＡ７が割り当てられ，どちらが使われるかは，プログラム実行状態，すなわち，Ｓビットによって自動的に決定されてしまうため，Ａ０～Ａ６までのいずれかのアドレスレジスタを経由しなければならないことを意味します．

【タスクの推移に関する命令群】

RESET

本命令を実行すると68000のリセット端子がアサートされるので，リセット端子に接続されているすべての外部デバイスをハード的にリセットできます．ただし，特別にリセット例外処理が起動されるわけではなく，PC を除きプロセッサの状態には影響を与えず，プログラムの実行は，次の命令から継続して実行されます．

RTE

RT は Return，E は Exception を意味し，例外処理からの復帰命令です．これは，例外処理の入口で退避しておいたシステム情報を取り出し，例外処理を発生させたプログラムへ制御を移行するものです．つまり，例外処理発生前の状態に復帰させる命令です．

たとえば，TRAP 命令でユーザプログラムからスーパーバイザへ分岐し，必要なサービスの終了後，ユーザプログラムへもどる時などに使用します．

STOP　#<data>

割り込み待ちをする命令で，割り込みが発生するまで MPU の動作を停止します．本命令は，マルチプロセッサ・システムを構築し，各プロセッサ間での同期をとるために用意されています．

たとえば，マスタ・プロセッサとスレーブ・プロセッサがあり，マスタ・プロセッサは，スレーブ・プロセッサへの仕事を割り込みで要求するとしましょう．スレーブ・プロセッサは，マスタからの割り込みで起動され，必要なサービスを終了後，STOP 命令を実行します．そして，次の「仕事の合図」，つまり，次の割り込みが発生するまでの間，事実上の停止状態を保持します．

オペランドに指定するイミディエイト値は，そのままSRへ転送されます．STOP 命令実行時にトレースビット（ビット15）がイチ（１）であると，本命令によってトレース例外処理が起動されますが，ほとんどの場合，マルチプロセッサ間の同期には不要であり，トレースビットをクリアしておく必要があります．

17 その他の命令[NOP]

この命令に分類される命令はNOPという何もしない命令です.

PCは本命令の次に位置している命令を継続して実行するために更新されますが，その他は，プロセッサの状態に影響を与えません.

何もしない命令が命令セットとして存在するからには，必要だからであり，以下のような用途を考えることができます.

① プログラムにパッチをあてる.
② デバッグに便利なことがある.
③ プログラム領域をNOPのコード(＄４Ｅ７１)で満たすことで，プログラム領域の予約をする.
④ マルチプロセッサ間の同期に応用する.

18 例外処理とは何か

68000は「例外処理」という処理と，「例外ではない」処理を行うことができますが，「例外である」とか「例外ではない」とは，何を意味するのでしょうか．

例外処理に関する知識は，アプリケーション・プログラムだけの開発しかしないのであれば，おそらく要求されることはないでしょうが，68000を根底から使いこなす上では，必修事項となります．

通常のプログラムはユーザ状態で走っており，この間何事もなく無事であってくれればよいのですが，そうではない(正常ではない事態の発生)こともあり得るわけです．

そこで，異常事態発生の可能性が存在するのであれば，それなりの対処をしようではないか，という〝単純な疑問〞に対する解答が例外処理の考え方です．

従来のマイクロプロセッサでは，異常事態は発生しないものとして設計がなされているので，プログラムの暴走からシステム・ダウンしても，それはプログラムを作成した人やそれを使った人が悪いのだ，ということになり，コンピュータを操作する人もプログラマも，「コンピュータだから」ということで納得しているのが現状です．

これは従来の8ビットプロセッサや8086に代表される16ビットプロセッサでは危険がある，という意味ではなく，現在のLSI製造技術から，マイクロプロセッサやその周辺LSIおよびメモリ素子などの各種半導体は，規格を無視した使い方をしない限り，少なくともテストランに成功すればエラーは発生しないものである，と断言できますから，最後には，プログラムの完成度だけが残ることになります．

どのようなプロセッサであっても，プログラムの完成度そのものが議論されるべきであり，「開発過程にある未完成のプログラムによる暴走は仕方ない」，ということなのですが，例外処理がサポートされた環境では，たとえプログラムに問題があってもシステム・ダウンせず，従来のマイクロプロセッサとは明らかに次元が異なります．

本セクションの内容を的確に把握するため，解説されているような機能が存在しない場合のことを考えてみてください．「例外処理」という処理のもつ重要さ，68000のすばらしさを理解できるはずです．

本セクションの解説ですが，68000にはどのような処理状態があるのか，特権状態とは何を意味するのか，ハード的にメモリ管理はどのように行われるのか，という事柄にふれ，最後に例外処理プログラムを開発するために必要な解説を行っています．

19 処理状態

［1］ 処理状態

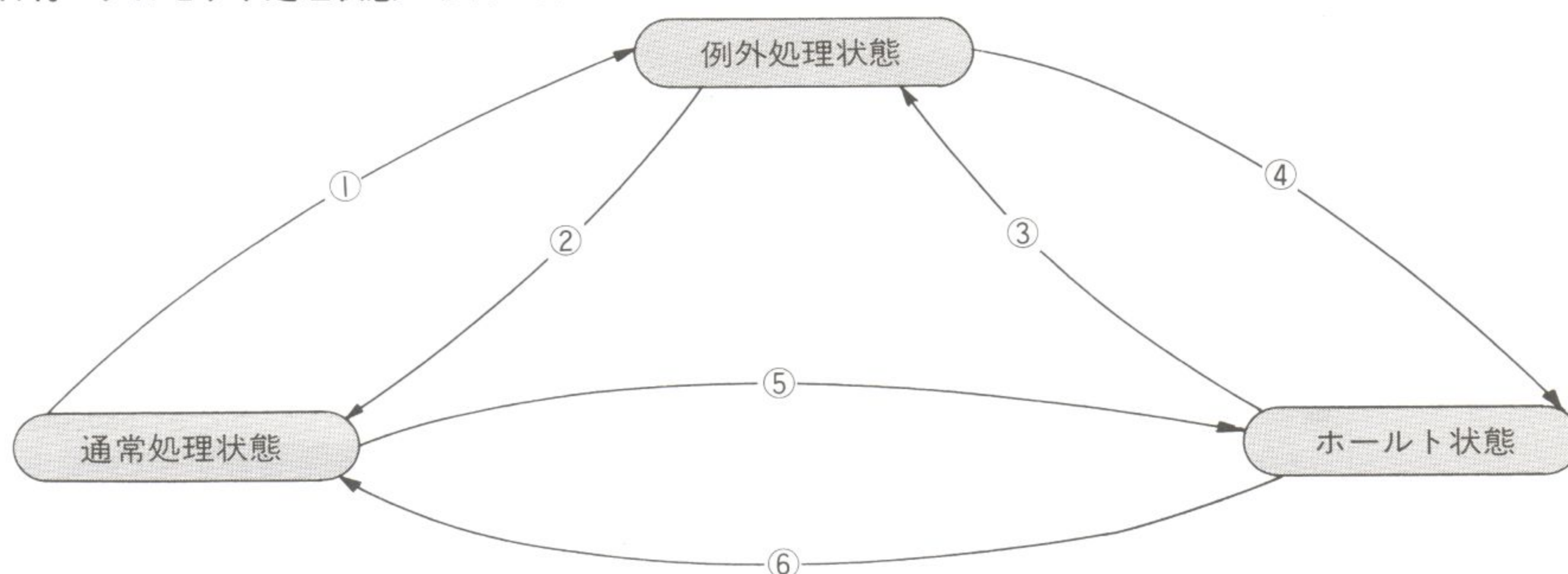
通常状態では，68000は適当なメモリ領域からプログラムを読み込み，それを解読し，プログラムを実行しますが，これ以外に例外処理状態やホールト状態が存在し，３つの処理状態が存在します．

以下，これら３つの処理状態を**表1.24**に整理します．また，その様子を**図1.46**に示します．

表1.24　３つの処理状態

処理状態	説明	
通常の処理状態	メモリを参照し続けるか否かで２つに分類されるが，これはSTOP命令を実行したかどうかという問題でもある．	
	●STOP命令を実行しない	プログラムを実行し続けるためにメモリを参照し続ける状態で，これが一般的な通常の状態である．
	●STOP命令を実行	通常状態の特殊なケースとして，STOP命令を実行すると68000は停止状態となり，これ以上はメモリを参照しないが，これも通常の処理状態に分類される．
例外処理状態	例外処理状態は他の処理状態ではないわけだから，この処理状態へ移行するには幾つかの要因がある．	
	●内部要因	命令あるいは命令実行の際に発生する異常事態による．
	●外部要因	割り込み，バスエラー，リセットによる．
ホールト状態	この状態も通常とは異なり，ホールト状態に移行する要因が２つある．	
	●$\overline{\text{HALT}}$端子(入力)を外部からアサートする	外部スイッチで$\overline{\text{HALT}}$端子を操作し，ハード的にシングル・ステップ動作を意図したもので，通常状態または例外状態で，$\overline{\text{HALT}}$端子がアサート(アクティブになる)されると，ホールト状態に移行し，その後$\overline{\text{HALT}}$端子をネゲート(インアクティブ)すると，ホールト状態に移行する前の状態に復帰する．
	●２重バス障害による	このケースではバスエラー・エクセプションが発生していることが前提になる． バスエラー・エクセプションによって例外処理状態へ移行し，ここで再びバスエラーが発生することを２重バス障害と呼び，68000はホールトする． １つ目の要因ではホールト状態から脱出できたが，本ケースではハード的にリセットする以外に，68000を再起動できない． この異常事態を外部へ知らせる為，68000は$\overline{\text{HALT}}$端子(出力)をアサートする．

図1.46 プロセッサ処理状態の移行の様子

例外処理状態
通常処理状態
ホールト状態
①②③④⑤⑥

①通常処理状態 → 例外処理状態
- $\overline{\text{RESET}}$端子をアサート
- 外部割り込みの受け付け
- TRAP命令の実行
- 不当命令、未実装命令の実行
- 特権違反
- トレース
- バス・エラー
- アドレス・エラー

②例外処理状態 → 通常処理状態
- 例外処理の終了

③ホールト状態 → 例外処理状態
- $\overline{\text{RESET}}$端子をアサート
- $\overline{\text{HALT}}$端子をネゲート

④例外処理状態 → ホールト状態
- $\overline{\text{HALT}}$端子をアサート
- 2重バス障害

⑤通常処理状態 → ホールト状態
- $\overline{\text{HALT}}$端子をアサート

⑥ホールト状態 → 通常状態
- $\overline{\text{HALT}}$端子をネゲート

[2] 特権状態とメモリの参照分類，および特権状態の変更

特権状態には上位の特権状態と下位の特権状態が存在し，いずれもプログラムの走行環境に関する問題であり，上位の特権状態が真の意味での特権状態を意味し，下位の特権状態では，特権化された命令(命令セットの幾つかは特権化されている)を実行できないように管理されています．

特権状態は，コンピュータ・システムの信頼性を向上させるためにあり，下位階層に位置する大部分の命令セットと，上位階層に位置する特権化された命令セット間で明確にメモリ空間が区別され，通常のプログラムによって参照あるいは変更される領域と，上位階層だけが参照あるいは変更できる領域とが存在し，システムにとって大切なメモリ空間を，下位階層の命令による破壊から保護することができます．

特権命令は極めて特殊な命令であり，システムの信頼性を支えるための命令なので(ユーザ状態での命令セットが従来の命令セットである)，「使える命令が限定されている」と判断しないでいただきたく，ユーザ状態での命令セットで，すべてのプログラム開発を行うことができる，と解釈してよいでしょう．

ここでのポイントは，命令には階層が存在すること，メモリ空間もまた命令の階層（特権状態）に応じて明確に管理されていることです．

特権状態の整理

表1.25は，特権状態について整理したものです．

表1.25　特権状態

特権状態	意　　味
上位階層の状態	●ステータスレジスタ(SR)のSビットがアサート（有効，〝1〞）されているとき，68000はスーパバイザ状態でプログラムを実行していることになり，特権命令を含むすべての命令セットを実行できる． ●メモリの参照はスーパバイザ参照であり，結果的にシステムスタックが操作される命令では，SSP（スーパバイザ・スタックポインタ，A７が割り当てられる）が用いられる． ●ＳＲのＳビットの状態にかかわらず，すべての例外処理（エクセプション処理）はスーパバイザ状態で実行され，メモリ参照はスーパバイザ参照である（もちろん，システムスタックはSSPで管理される）．
下位階層の状態	●ステータスレジスタ（ＳＲ）のＳビットがネゲート（無効，〝0〞）されているとき，68000はユーザ状態でプログラムを実行していることになり，命令セット中の特権命令は実行できず，もし実行すれば，特権命令違反として例外処理が起動される． ●メモリの参照はユーザ参照であり，結果的にシステムスタックが操作される命令では，USP（ユーザ・スタックポインタ，A７が割り当てられる）が用いられる．

note：システム・スタックポインタは特権状態の各階層ごとに存在し，同じＡ７に割り当てられるが，ユーザ状態のＡ７とスーパバイザ状態のＡ７とは別々のものである．

参照分類

ＳＲのＳビットがアサートされているか否かによって，どのメモリ領域への参照なのかを外部へ知らせるための信号(FC_2，FC_1，FC_0)があり，この信号をデコードしてはじめて特権命令がその役割を果たすことができます．

メモリ空間には，ユーザ空間とスーパバイザ空間が存在しますが，さらに各空間は，プログラムが位置している空間と，プログラム領域ではない単なるデータ空間，の２つが存在しますから，４通りのメモリ空間が存在します．

現状に注目してみますと，たとえこのようにメモリ空間が分離管理されているとはいえ，ハードウェアによるワイヤード・ロジックでは，メモリを効果的に割り当てることは不可能であり，高度なオペレーティング・システム（ＯＳ）では，メモリ・マネージメント・ユニット(MMU)と呼ばれるLSIを操作することによって，メモリをダイナミックに管理することが行われています(FC_2，FC_1，FC_0はMMUへインターフェースされる)．

ハード的に４つのメモリ空間の区別はFC_2，FC_1，FC_0という名称の端子から出力される状態で判別でき，この様子を**表1.26**に示します．

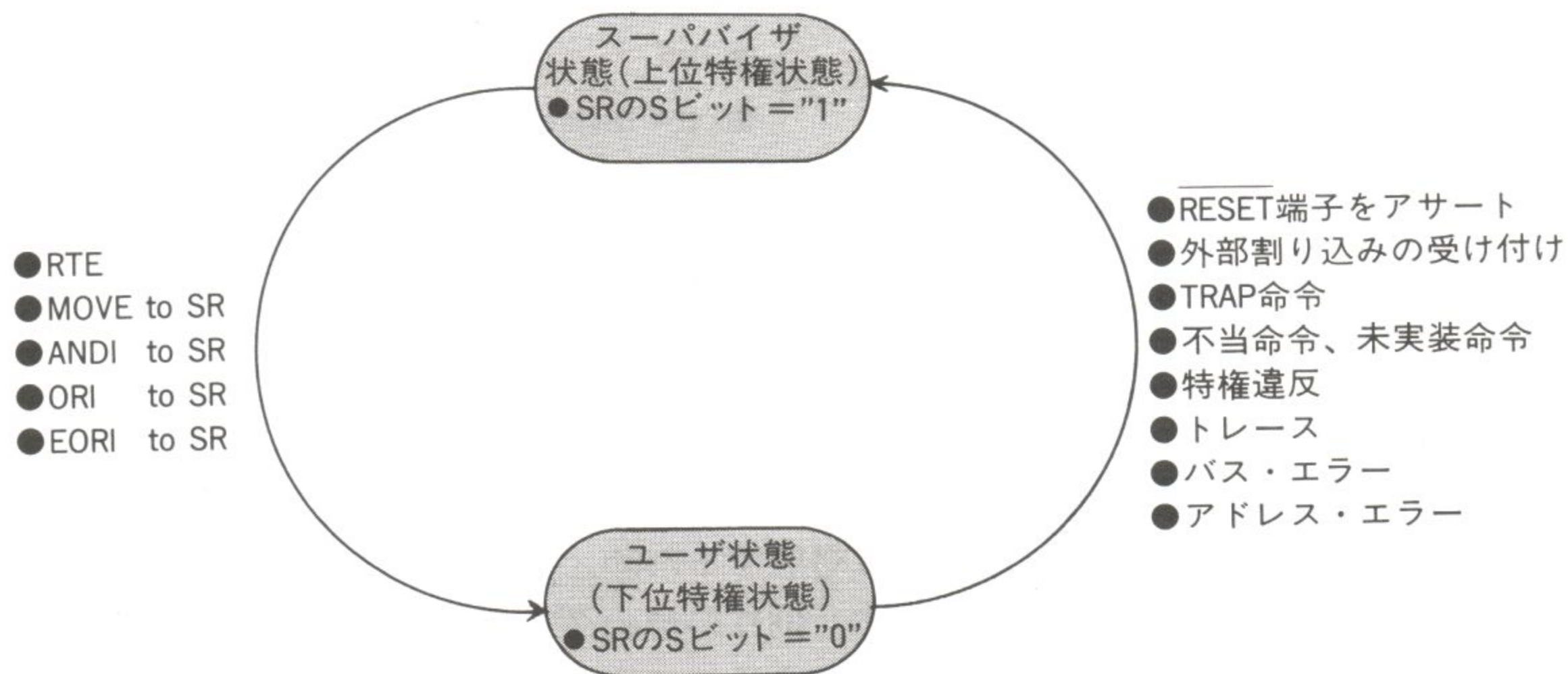
表1.26
参照の分類

FC(Function Code)			参照されるメモリ領域
FC_2	FC_1	FC_0	
0	0	0	割り当てなし(未定義)
0	0	1	ユーザ・データ領域
0	1	0	ユーザ・プログラム領域
0	1	1	割り当てなし(未定義)
1	0	0	割り当てなし(未定義)
1	0	1	スーパバイザ・データ領域
1	1	0	スーパバイザ・プログラム領域
1	1	1	割り込みアクノリッジ

特権状態の変更

68000の内部でどのようなことが起こっているかという問題は後述するとして，通常はユーザ状態で一連のプログラムが実行され，以後，例外処理だけが特権状態を変更でき，スーパバイザ状態からユーザ状態への復帰は，RTE(例外処理からの復帰)命令や，ＳＲのＳビットを直接操作する命令で実現できます．

この様子を**図1.47**に示します．

図1.47　特権状態移行の様子

20 例外(エクセプション)処理

例外処理プログラムを開発するための具体的な解説を行いますが，読者諸氏が困惑しないように，「何をどう操作すればよいのか」という点に重点をおいた解説をしています．

［1］ 例外処理の分類とその優先度

例外処理が起動される要因には内部要因と外部要因とがあり，同時に複数の例外処理（多重例外と呼ぶ）が起動された場合に対処するため，例外処理には優先順位が与えられています．

例外処理の要因とその種類

例外処理の要因とその種類との関係を**表1.27**に示します．

表1.27 例外処理の要因とその種類

例外処理を起動する要因	起動される例外処理の種類
外部要因	●ハードウェア・リセット ●ハードウェア割り込み ●バス・エラー
内部要因	●TRAP　命令 ●TRAPV　命令（CCRのＶビットによる） ●CHK　命令（結果による） ●Zero Divide　（０による除算時）
	●不当命令，未実装命令 ●特権違反
	●トレース処理
	●アドレス・エラー

例外処理の優先度と起動時のタイミング

複数の例外処理が同時に起動された場合に対処するため，例外処理には優先度が存在し，その優先度に従って例外処理が起動されます．例外処理のグループ分け，優先度，種類，起動時のタイミング，などの関係は**表1.28**に示す通りです．

例外処理の検出タイミング

68000がどのようにして例外処理の有無を判断しているかを以下に整理しますが，大切なことは，例外処理が起動されサービス・プログラムへ制御が移行したとき，68000がスタックへ退避するＰＣの内容は何を意味するか，ということです(次ページ**表1.29**)．

表1.28　例外処理の優先度と起動時のタイミング

優先度	例外処理の種類	例外処理が起動(実行)されるタイミング
最高位(Group 0)	0：ハードウェア・リセット 1：アドレス・エラー 2：バス・エラー	2クロックサイクル以内に起動される (次のマシン・サイクルで起動)
中位　(Group 1)	3：トレース 4：ハードウェア割り込み 5：不当命令，未実装命令 5：特権命令	次に位置する命令より先に起動される
最低位(Group 2)	6：TRAP, TRAPV 6：CHK 6：Zero Divide	通常の命令実行によって起動される (その命令の実行途中で起動)

note：①各グループ間ではグループ0が最も優先度が高く，グループ2の優先度が最低位である．
②例外処理の種類欄の先頭に付加されている〝0：〞～〝6：〞は，全体的な優先度を意味し，〝0：〞が最高位で〝6：〞が最低位となる．
③グループ0，1に関しては同じグループ内でも優先度が存在するが，Group 1の〝5：〞やグループ2はソフトウエアに依存するので同じ優先度である．たとえば，TRAP命令とCHK命令は同時に実行できない，等．

表1.29　例外処理の検出タイミング

Group	例外処理の種類	検出のタイミング
0	ハードウェア・リセット	クロック・サイクルの最後
	アドレス・エラー バス・エラー	クロック・サイクルごと
1	トレース	バス・サイクル（命令サイクル）の最後
	ハードウェア割り込み 不当命令，未実装命令 特権命令	実行中の命令サイクルの最後
2	TRAP，TRAPV CHK Zero devide	命令サイクル内

［2］　例外ベクタ

例外処理の解説を明瞭にするため，例外ベクタについての解説を前置きします．

例外ベクタとは，例外処理ルーチンの分岐先が記憶されている特別なメモリ・ロケーションのことで，ロングワード(4バイト)を基本単位とします．とにかく，例外処理をするために，68000はこのような特別のメモリ空間を必要とします．

例外ベクタ

68000に電源が投入され，外部ロジックで作成されたハードウェア・リセット信号が有効になると($\overline{\text{RESET}}$，$\overline{\text{HALT}}$端子をアサート)，68000はリセット例外処理を実行します．このとき例外ベクタは，次のように参照されます．

① ゼロ番地から連続する4バイトのエリアに格納されている内容をSSPに取り込む(SSPの初期化).
② 4番地から連続する4バイトのエリアに格納されている内容をＰＣに取り込む(ＰＣを初期化).

「何々番地から連続する4バイトのエリア」ではスマートな解説ではないので，例外ベクタ番号や例外ベクタ，あるいはベクタ番号とか単にベクタという表現をすれば，

ベクタ番号0(ゼロ)の内容をSSPに取り込む.

と説明できます.

場合により，リセット例外処理が起動されたときに参照されるベクタ番号を〝リセット・ベクタ″と表現し，起動される例外処理の名称とそのときに参照されるベクタ番号とは，原則として1対1で対応し，この様子は他の例外処理に対するベクタ番号にも適用できます.

例外ベクタの割り当て

例外ベクタの割り当てについては，**表1.30**を参照してください．その内容を以下に要約します.

① ベクタ番号は0～255までの256通りで，各ベクタの内容は4バイトであることから，ベクタ領域は1024(256＊4)バイトである.
② ベクタ番号0(ゼロ)の先頭は物理アドレスのゼロ番地に，ベクタ番号255の先頭は物理アドレスの＄3FCに位置する.
③ リセット・ベクタだけはベクタ番号0と1の2つを必要とするので，ベクタ番号1が存在しないような表記になっている.
④ ハードウェア割り込みでは，割り込みを要求した周辺LSIがベクタ番号を出力する約束になっているので，68000自身がベクタ番号を決定するわけではない.
⑤ ユーザ割り込みベクタとして，ベクタ番号64～255までの192個が確保されている.

[3] 例外処理シーケンス

例外処理シーケンスには4つのステップがあり，各ステップについて解説しますが，フローチャートにはそのすべてが集約されていることから，こちらの方を常に参照すべきです(**図1.48～図1.51**).

例外処理といっても，最終的にはそれなりの処理ルーチンへ分岐するわけで，以下の項目に注目するとよいでしょう.

① 例外処理の過程で68000自身が操作するレジスタは何か.
② 処理ルーチンはどのようにして決定されるのか.
③ 処理ルーチンへ分岐してきたときの状態はどうなっているのか.
④ どのようにして例外処理を脱出すればよいか.

というわけで，68000の内部レジスタの推移やスタックの様子を中心に追っていけばよいことになります．**表1.31**に例外処理シーケンスを整理します.

表1.30　例外ベクタの割当て

SP：Supervisor Program
SD：Supervisor Data

ベクタ番号	ベクタの先頭アドレス			割り当ての内容
	10進	16進	メモリ空間	
0	0	$000	SP	リセット：SSPの初期値
1	4	$004	SP	リセット：PCの初期値
2	8	$008	SD	バス・エラー
3	12	$00C	SD	アドレス・エラー
4	16	$010	SD	不当命令
5	20	$014	SD	0による除算
6	24	$018	SD	CHK命令
7	28	$01C	SD	TRAPV命令
8	32	$020	SD	特権違反
9	36	$024	SD	トレース例外処理
10	40	$028	SD	未実装命令（line　1010 emulator）
11	44	$02C	SD	未実装命令（line　1111 emulator）
12*	48	$030	SD	割り当てられていない（予約されている）
13*	52	$034	SD	割り当てられていない（予約されている）
14*	56	$038	SD	割り当てられていない（予約されている）
15	60	$03C	SD	アンイニシャライズド割り込み
16* ～ 23*	64 ～ 92	$040 ～ $05C	SD	割り当てられていない（予約されている）
24	96	$060	SD	スプリアス割り込み
25	100	$064	SD	レベル1割り込み，オート・ベクタ方式
26	104	$068	SD	レベル2割り込み，オート・ベクタ方式
27	108	$06C	SD	レベル3割り込み，オート・ベクタ方式
28	112	$070	SD	レベル4割り込み，オート・ベクタ方式
29	116	$074	SD	レベル5割り込み，オート・ベクタ方式
30	120	$078	SD	レベル6割り込み，オート・ベクタ方式
31	124	$07C	SD	レベル7割り込み，オート・ベクタ方式
32 ～ 47	128 ～ 188	$080 ～ $0BC	SD	TRAP　#0 命令～TRAP　#15　命令
48* ～ 63*	192 ～ 252	$0C0 ～ $0FC	SD	割り当てられていない（予約されている）
64 ～ 255	256 ～ 1020	$100 ～ $3FC	SD	ユーザ割り込みベクタ（192種類）

＊これらのベクタ番号は将来使用される可能性があるので，ユーザが使用すべきでない．

表1.31　例外処理のステップ

例外処理のステップ	説　　明
(1)	SRの退避とS，T，I_2，I_1，I_0などのＳＲビットを操作． ①　SRを内部的にコピー(退避)する． ②　スーパバイザ状態にするため，ＳＲのＳビットをアサートし，例外処理をトレース処理から保護するため，ＳＲのＴビットをネゲートする． ▶　リセット例外処理や割り込み例外処理では，ＳＲのI_2，I_1，I_0を操作し，割り込み優先マスクも更新する．
(2)	例外ベクタ番号を決定し，これを４倍してベクタアドレスを求めるが，ベクタ番号の決定には２つのケースがある． ▶　ハードウェア割り込みによる場合，割り込みを要求した外部ディバイスから送り出されるベクタ番号を受けとる． (割り込みアクノリッジ) ▶　ハードウェア割り込みではない場合，内部ロジックによりベクタ番号を生成する．
(3)	リセット例外を除き，現在のＰＣ(プログラム・カウンタ)とＳＲ(ステータス・レジスタ)をスーパバイザ・スタック・エリアへ退避する． この結果SSPの値は例外開始時の値から６だけ減じられている． ①　ＰＣ(４バイト)をスーパバイザ・スタック・エリアへプッシュ． ②　例外処理時の入り口で内部に退避しておいたＳＲ(２バイト)をスーパバイザ・スタック・エリアへプッシュ．
(4)	例外処理の処理先アドレスをベクタ領域から取り込み，そこへ制御を移行するが，この様子はすべての例外処理に共通である． 制御を移行するということはＰＣの内容が書き換わることであり，すでにベクタ番号からベクタ・アドレスは求められているので，そのロケーションの内容をＰＣへ取り込んで，指定ルーチンへ分岐する．

〈NOTE〉

スーパバイザ・スタック空間

	15　　　　　　　　　　0
下位アドレス	
SSP→	例外処理発生前のＳＲ
	ＰＣの上位ワード
	ＰＣの下位ワード
上位アドレス	

スタックされたＰＣの値は，通常，次のまだ実行されていない命令，すなわち，例外処理が起動されなかったときに実行すべき命令が格納されているアドレスをポイントしているが，バスエラーとアドレスエラーに対しては，エラーを発生(例外処理の起動要因)させた命令のアドレスに対して，インクリメントされている可能性もある．

バス・エラー例外処理とアドレス・エラー例外処理の場合，現在のプロセッサの内容を示す追加情報が，更にスーパバイザ・スタック・エリアにプッシュされる．

図1.48　例外処理のフローチャート●1

リセット例外処理

1 → S　　スーパバイザ状態
0 → T　　トレース処理禁止
1、1、1 → I_2、I_1、I_0　　割り込みレベル7

＊ベクタ番号0の内容を取り込む

バス・エラーか？　（YES → 2重バス障害とみなしHALTする）

NO

取り込んだ内容→SSP　　SSP初期化

＊ベクタ番号1の内容を取り込む

バス・エラーか？　（YES → 2重バス障害とみなしHALTする）

NO

取り込んだ内容→PC　　PC初期化

PCの内容を取り込む（オペレーション・コード・フェッチ）

アドレス・エラー or バス・エラーか？　（YES → 2重バス障害とみなしHALTする）

NO

PC＋2の内容を取り込む（次のオペレーション・コード・フェッチ）

バス・エラーか？　（YES → バス・エラー例外処理を起動）

NO

リセット例外処理終了

システム起動プログラムを実行

2重バス障害とみなしHALTする

バス・エラー例外処理を起動

＊ベクタ番号0とかベクタ番号1の内容とは、ベクタ番号を4倍して得られたベクタ・アドレスの内容のことで、それぞれ0番地、4番地に格納されているロングワード・データを意味する

図1.49　例外処理のフローチャート● 2

バス・エラーまたは
アドレス・エラー例外処理

SRを68000内にコピー

1 → S
0 → T

●バス・エラーの場合　　：ベクタ番号 2 を内部で発生
●アドレス・エラーの場合：ベクタ番号 3 を内部で発生

ベクタ番号を 4 倍してベクタ・アドレスを得る

①PCの下位ワードをスーパバイザ・スタックへPUSH
②PCの上位ワードをスーパバイザ・スタックへPUSH
③内部にコピーしておいたSRをスーパバイザ・スタックへPUSH
④例外処理を発生させた命令のオペレーション・ワードをスーパバイザ・スタックへPUSH
⑤例外処理を発生させたメモリ・アドレスの下位ワードをスーパバイザ・スタックへPUSH
⑥例外処理を発生させたメモリ・アドレスの上位ワードをスーパバイザ・スタックへPUSH
⑦例外処理を発生させたメモリ・アクセスのアクセス情報をスーパバイザ・スタックへPUSH

アドレス・エラー or バス・エラー か?　YES / NO

ベクタ・アドレスの内容(例外ベクタとも言う)を取り込む

アドレス・エラー or バス・エラー か?　YES / NO

例外ベクタをPCにセット

オブジェクト・コードの最初のワードを取り込む

アドレス・エラー or バス・エラー か?　YES / NO

オブジェクト・コードのその次のワードを取り込む

アドレス・エラー or バス・エラー か?　YES / NO

バス・エラーまたは
アドレス・エラー例外処理完了

処理プログラムの実行

2 重バス障害とみなし
ホールト状態になる

図1.50　例外処理のフローチャート● 3

割り込み例外処理

SRを68000内にコピー

1→S
0→T
割り込みレベル→I_2〜I_0

PCの下位ワードをスーパーバイザ・スタックへPUSH

アドレス・エラー or バス・エラー ?　YES / NO

割り込みアクノリッジ：
● 割り込みレベルをアドレスバスA_3〜A_1に出力
● R/W端子をHigh
● FC_2〜FC_0端子をすべてHigh
● $\overline{AS}$、$\overline{UDS}$、$\overline{LDS}$端子をすべてアサート

$\overline{VPA}$アサート？　YES / NO

オート・ベクタ：
割り込みレベルに応じたベクタ番号（25〜31）を内部で発生

外部ディバイスから送り出されるベクタ番号（64〜255）を取り込む

バス・エラー？　YES / NO

スプリアス割り込み：
ベクタ番号24を内部で発生

ベクタ番号を4倍してベクタ・アドレスを得る

①PCの上位ワードをスーパバイザ・スタックへPUSH
②コピーしておいたSRをスーパバイザ・スタックへPUSH

アドレス・エラー or バス・エラー か？　YES / NO

ベクタ・アドレスの内容（例外ベクタとも言う）を取り込む

アドレス・エラー or バス・エラー か？　YES / NO

例外ベクタをPCへセット

オブジェクト・コードの最初のワードを取り込む

アドレス・エラー or バス・エラー か？　YES / NO

オブジェクト・コードのその次のワードを取り込む

アドレス・エラー or バス・エラー か？　YES / NO

割り込み例外処理完了

割り込み例外処理サービス・プログラムを実行

割り込み例外処理完了

アドレス・エラー　または バス・エラー例外処理開始

図1.51　例外処理のフローチャート● 4

21 例外処理の詳細

システム・エンジニアにとって，例外処理は非常に重要であります．
本セクションでは，その過程を的確に解析し，プログラマは「何をどう操作すればよいか」，という問題点を的確に把握できるような解説をします(必要に応じてこれまでの解説を参照してください)．

1 ハードウェア・リセットによる例外処理

●ベクタ番号：〈0〉,〈1〉　●優先度：［Group 0］

要点

ハードウェア・リセット例外処理は，68000に電源が投入されたときやコンピュータ・システムのリセット・スイッチを操作した場合に，どのようにシステムを起動したらよいか，ということであり，スタートまたはリスタート処理を行うものである．
リセット例外処理後，その先でどのようなプログラムを実行するかは，プログラム開発マシンなのか，I/Oプロセッサとして設計されているのかなど，68000がどのような目的で使われているかに依存する．

■SSPとPCの内容は，それぞれベクタ0と1の内容が取り込まれるので，それぞれ0番地からの4バイトと4番地からの4バイトに，必要な値を設定しておけばよい．

■ハードウェア・リセットによって無事に所定処理ルーチンへ分岐したら，システムに必要な処理を行い，USP（ユーザ・スタック・ポインタ）などユーザ状態のプログラム走行環境を設定後，MOVE/ANDI/ORI/EORI to SR命令のいずれかを実行してユーザ状態へ制御を移行できるが，RTE命令を効果的に使用することも重要である．

■システムの初期化は必要であるが，実行すべきプログラムを他のシステムから受け取って，それに対するサービスだけを目的とする場合もある．

要因

ハードウェア・リセットによって68000がリセットされたとき．

68000の応答

1：　SRのS，T，I_2～I_0，の各ビットを操作する．

SR	ステータス	設定される環境
S	1	スーパバイザ状態
T	0	トレース処理の禁止
I_2，I_1，I_0	1，1，1	割り込みレベル7(後述)

2：　以下のようにしてプログラムが起動される．
0番地からの4バイト（ベクタ0）をSSPへ取り込む（SSPの初期化）．
4番地からの4バイト（ベクタ1）をPCへ取り込む（PCの初期化）．

注意事項

■スタート／リスタートという意味

本例外処理は最高優先度の例外処理で，ハードウェア・リセットが発生すると，もし進行中の処理があっても中断され，回復することはできず，〝どのような状態であったか〟ということは無視される．

■リセット例外処理中のバスエラー／アドレスエラーの発生

68000はメモリをアクセスする場合には，バスエラーまたはアドレスエラーが発生したか否かを常にチェックしており，例外処理中に発生すると２重バス障害とみなし，ホールトする．

リセット例外処理でホールトするケースは，ベクタの取り込みに失敗したとき，ＰＣで指定されるアドレスの内容（最初のオブジェクト・ワード）を取り込めなかったときである．ただし，ＰＣ＋２で指定されるアドレスの内容が取り込めなかった場合は，バスエラー例外処理が起動され，ホールトしない．

いったんホールト状態になると，以後再びハード的にリセットしなければ68000を２度と起動できないが，コンピュータシステムのリセットスイッチを押すか，あるいは電源を再び投入しても同様な結果となる可能性が大であるから，メモリのインターフェースを再検討する必要がある．

■RESET命令

RESET命令は68000のリセットラインをアサートするだけで，本ラインにインターフェースされている周辺ディバイスをソフト的にリセットすることを目的とし，リセット例外処理は起動されない（RESET命令は特権命令である）．

2 バスエラー例外処理

●ベクタ番号：〈2〉 ●優先度：［Group 0］

概要

割り込み処理などは小規模システムでは不要なこともあるが，バスエラー例外処理はシステムの大小を問わずサポートされるべきもので，68000の例外処理ではきわめて一般的なものである．

どのような場合に本例外処理を要求するかは，外部ロジックがどのようなケースをバスエラーと判断するかに依存するが，プロセッサは$\overline{\text{BERR}}$端子がアサートされると，命令実行中であろうと他の例外処理中であろうと，現在のバスサイクルを中断し，ベクタ番号2を内部で発生し，所定処理ルーチンへ制御を移行する．

本例外処理はかなり致命的な部類に入るので，完全にシステムを回復するのは困難であるが，システムプログラムやプログラムのデバッグ中にデバッガが破壊されることはなく，従来のプロセッサのように，プログラムの異常からシステムダウンすることはない．

要点

本例外処理はいたるところでトラップされる可能性があり，完全にシステムを回復することは困難かもしれない．ただしプログラム自体のモジュール化や階層化を徹底すれば，エラーからの回復チャンスも生じるので，システム全体をリセットせず，適当であると判断したロケーションへ制御を移行できるかもしれない．

バスエラー例外処理を行うサービスプログラムへ制御が移行したとき，ＰＣやＳＲ以外にもシステムを回復する情報がスタックされているので，回復できるエラーであれば，スタックをクリーンアップしてからリターンしなければならない（途中でスーパバイザ・スタックへ退避される情報が多い）．

要因

バスエラー例外処理を起動するには，$\overline{\text{BERR}}$という端子を外部からアサートする必要があるが，ハードウェア設計者は以下のような要因を考慮している．

- ■未実装のメモリ空間やＩ/Ｏディバイスをアクセスし，結果としてこれらのディバイスが応答しない場合．
- ■メモリエラーが発生した場合．
- ■ユーザ状態でスーパバイザ空間をアクセスした場合が典型的な例であるが，MMU（メモリ・マネージメント・ユニット）が不正なアクセス要求を検出した場合．
- ■ハードウェア設計者の立場から，トラップが必要であると思われるエラーに対処するため．

68000の応答

1：　ＳＲをコピーし，スーパバイザ状態に入り，トレースを禁止する．
2：　ベクタ番号2を内部で発生し4倍（ベクタ・アドレスを得る）する．
3：　以下の内容をスーパバイザ・スタックへプッシュするが，処理プログラム内でこれらの内容を解析すれば，どのようにしてシステムを回復すればよいか，という有力な解決手段となる．

■ＰＣ（プログラム・カウンタ）.　　<NOTE　1>

■本例外処理起動前のＳＲ.

■本例外処理を発生させた命令のオペレーション・ワード，すなわち処理中だった命令の第１オペレーション・ワード.

■中断されたバス・サイクルによってアクセスされつつあったアドレス.

■バスエラー時のアクセスに関する特殊情報.

- リードとライトのどちらなのか.
- 「命令実行中」なのかそうでないときであったのか.　　<NOTE　2>
- ファンクション・コード（FC_2～FC_0）の出力分類.

4：　バスエラー・ベクタ・アドレスの内容をＰＣへ読み込み，そこへ制御を移行する.
<NOTE　3>

注意事項　ハードウェア・リセット，バスエラー，アドレスエラー，などの例外処理中にバスエラーが発生すると，68000は２重バス障害とみなしホールトする．ホールトした68000を再び起動するには，電源を再び投入するかハードウェア・リセットする以外にない.

NOTE

<NOTE　1>　スタックされたＰＣの意味

この内容はバスエラーを発生した命令の第１ワードが格納されているアドレスから，数バイト先に進んでいることも予想される.

次の命令を取り込んでいる間にバスエラーが発生した場合，たとえその命令がブランチ命令，ジャプ命令，リターン命令であったとしても，スタックへ退避されたＰＣの内容は，現在の命令付近をポイントしていることも考えられる.

<NOTE　2>　「命令実行中」の分類

２番目の「命令実行中」とは，プロセッサが通常で命令を実行しているか，グループ２の例外処理を実行しているか，という分類であり，グループ０とグループ１の例外処理中は「命令実行中」に分類されない.

<NOTE　3>　例外処理の最終段階におけるＰＣの内容

例外処理の最後の段階で，例外ベクタを取り込んでいる間，または命令を取り込んでいる間にバスエラーが発生した場合，ＰＣの値は例外ベクタのアドレスとなる(バスエラーは常にトラップされることをフローチャート上で確認してほしい).

表1.33　アドレスエラーまたはバスエラー時のスタック

スーパバイザ・スタック

下位アドレスの方向

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ＳＳＰ→												＊1	＊2	＊3		
	例外処理の原因となったアクセスアドレス（上位ワード）															
	例外処理の原因となったアクセスアドレス（下位ワード）															
	例外処理の原因となった命令のオペレーション・ワード															
	例外処理が発生する前のＳＲ															
	ＰＣの上位ワード＊4															
	ＰＣの下位ワード＊4															

上位アドレスの方向

▶＊1　ビット4：　〝0〞——ライト・サイクルによって例外処理が発生した．
〝1〞——リード・サイクルによって例外処理が発生した．

▶＊2　ビット3：　〝0〞——命令をアクセスした．
〝1〞——命令以外をアクセスした．

▶＊3　ビット2，1，0：　FC_2，FC_1，FC_0（ファンクション・コード）の状態

▶＊4　ＰＣの値はバスエラーまたはアドレスエラーを発生させた命令の次の命令が格納されているアドレスを保持しているとは限らない．

▶　SSPは14（7ワード）だけ更新される

3 アドレスエラー

●ベクタ番号：〈3〉　●優先度：［Group 0］

概要

ワードオペランド，ロングワードオペランド，オブジェクトコード（命令）などは，必ず偶数番地から配置される約束なので．これらを奇数アドレスでアクセスした場合，アドレスエラー例外が発生する．

アドレスエラーは内部起因のバスエラー例外と考えてよく，そのときにどのような処理を実行していようとその処理を中止し，例外処理を開始する．

プログラム自体をトラップする性格が強いので，デバッガでは必ずサポートされなければならないし，もちろんシステムの信頼性向上にも重要な役割を果たす．

いうまでもなくこのようなエラーが発生するようなプログラムであれば，即座に修正しなければならない．

要点

アドレスエラーは68000自身がトラップするという相違はあるものの，例外処理開始後のシーケンスは，スタックされる情報を含めバスエラー例外処理と同様である．

要因

■ワードオペランドを奇数アドレスでアクセスしたとき．
■ロングワードオペランドを奇数アドレスでアクセスしたとき．
■命令（オブジェクトコード）を奇数アドレスから取り込んだとき．

68000の応答

バスエラー例外処理と同様．

4 ハードウェア割り込み例外処理

●ベクタ番号：〈64〉〜〈255〉　●優先度：［Group 1］

概　要　まず，以下の2事項に留意する必要がある．

■ハードウェア割り込みはきわめて重要であり，アセンブラでなくては記述できないものである．記述できる高級言語(C，FORTH)もあるが，記述できることとプログラムの完成度とは異質のものであり，効果的なデバッグも不可能である（アセンブラによるデバッグさえ十分な注意を要する）．

■単一レベルの割り込み処理ではなく，いずれつきあたる問題であろう「多重割り込み」に関する対策も大切である．

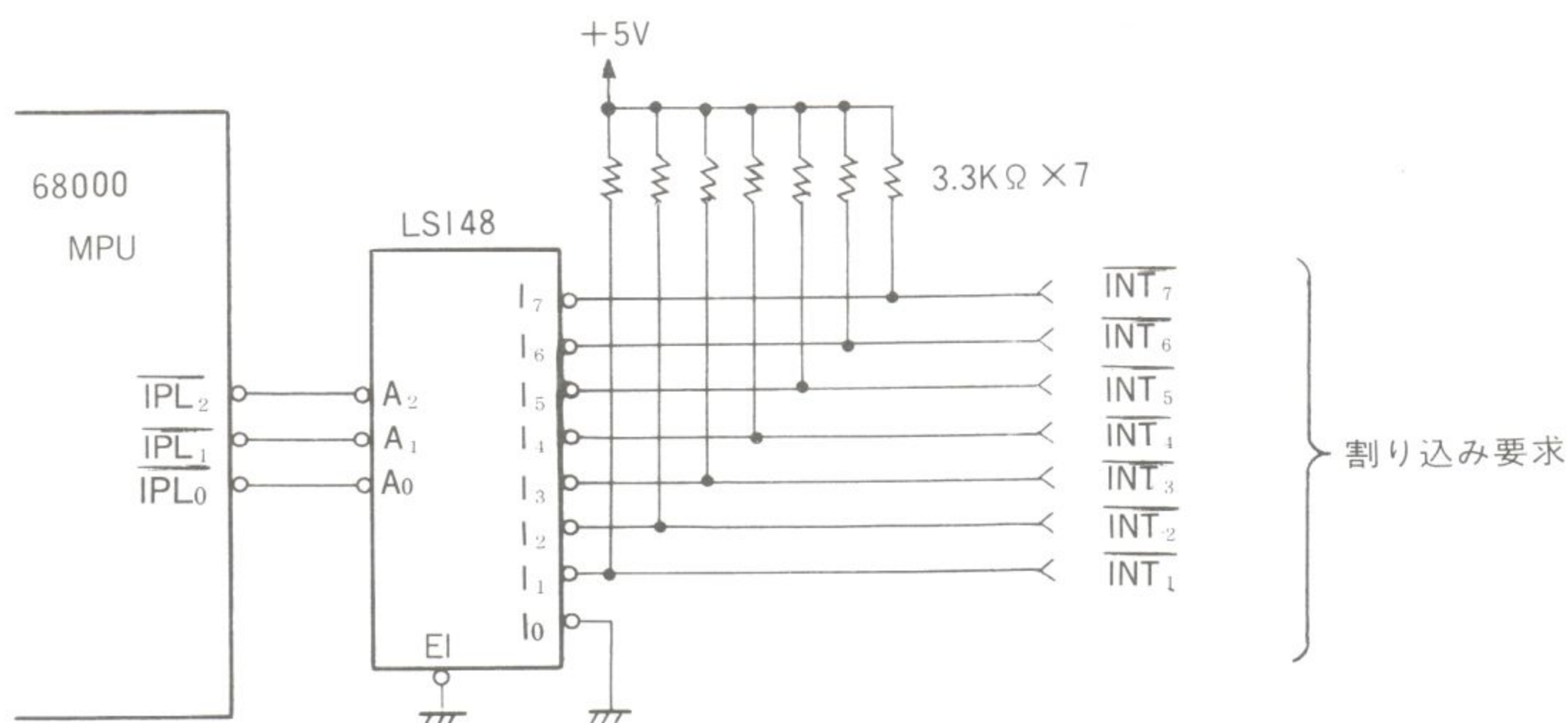
1．割り込み要求ラインと外部ロジック

68000には外部からの割り込み要求をセンスする端子として，3本($\overline{IPL_2}$，$\overline{IPL_1}$，$\overline{IPL_0}$)の割り込み制御入力端子があるが，単に$\overline{IPL_2}$，$\overline{IPL_1}$，$\overline{IPL_0}$へ割り込み要求信号をインターフェースするのではなく，〝割り込みレベル〞という考え方から，割り込み要求ラインのエンコード結果を$\overline{IPL_2}$，$\overline{IPL_1}$，$\overline{IPL_0}$の各端子へ与える．

割り込み制御端子（$\overline{IPL_2}$，$\overline{IPL_1}$，$\overline{IPL_0}$）は3ビットであるから，8本の割り込み要求を処理できるはずだが，$\overline{IPL_2}$，$\overline{IPL_1}$，$\overline{IPL_0}$がすべて〝High〞の状態は，〝割り込みなし〞を検出するために予約されているので，7本の割り込み要求端子となる．

こうして外部に実現された7本の割り込み要求ラインは，本来なら68000本体に用意されるべきものだが，プライオリティ・エンコーダを外付けすることで，端子数を節約しているわけである．

割り込み要求ラインの制約から，7個までのディバイスしかサービスできないわけではなく，1本の割り込み要求ライン上に，複数の外部ディバイスからの割り込み要求信号をインターフェースすることも可能なので，割り込みを要求する外部ディバイスの数に制限はない．

図1.52　外部割り込み要求信号の処理

2. 割り込みの優先順位と割り込みレベル

先述した7本の割り込み要求端子の様子は，外部で優先順を含めてエンコード(符号化)され，$\overline{IPL_2}$～$\overline{IPL_0}$ に与えられる．そしてこのコードは，あらかじめ I_2～I_0 にプログラムされたレベルと比較され，I_2～I_0のレベルより高いレベルであれば，現在実行中の命令が終了すると割り込み例外処理が起動され，同レベルかそれ以下ではマスクされる．

割り込みレベル間の優先度は，レベル7が最高位でレベル1が最低位，分類上レベル7はマスクできない割り込みで，他はソフトウェアでマスク(許可／禁止)可能である．また $\overline{IPL_2}$～$\overline{IPL_0}$ の各端子がすべてHighなら，68000は割り込み要求がないものと解釈する．

これまでの内容を**表1.34**，**表1.35**に整理するが，$\overline{IPL_2}$～$\overline{IPL_0}$ は負論理で I_2～I_0 は正論理なので，$\overline{IPL}$端子については電圧の状態を示すシンボルとして，L (Low) またはH (High) と表記している．

表34

エンコードの様子			発生する割り込み要求レベル	分　類
$\overline{IPL_2}$	$\overline{IPL_1}$	$\overline{IPL_0}$		
L	L	L	7	マスク不能
L	L	H	6	マスク可能
L	H	L	5	
L	H	H	4	
H	L	L	3	
H	L	H	2	
H	H	L	1	
H	H	H	割り込み要求なし（レベル0）	

（注）$\overline{IPL_2}$～$\overline{IPL_0}$ は負論理

表35

割り込みレベル	SR			意　味	
	I_2	I_1	I_0	マスクされるレベル	受けつけられるレベル
7	1	1	1	6～1	7
6	1	1	0	6～1	7
5	1	0	1	5～1	7,6
4	1	0	0	4～1	7,6,5
3	0	1	1	3～1	7,6,5,4
2	0	1	0	2～1	7,6,5,4,3
1	0	0	1	1	7,6,5,4,3,2
0	0	0	0	マスクしない	7,6,5,4,3,2,1

（注）I_2～I_0は正論理

3. 多重割り込み処理

割り込み処理は，コンピュータシステムを一層効果的にオペレートできるかどうか，という程重要な要素をもち，信頼性の高い割り込みシステムを実現するためには，慎重に外部ロジックを構成する必要がある．

3.1 多重割り込み処理という意味

現在のマイクロプロセッサの応用範囲を考慮すれば，割り込みを要求するディバイスが1つであることは考えられず，数箇所あるいはそれ以上の異なったディバイスからの割り込みを考慮しなければならない．つまり，多重割り込みにどう対処するのか，ということは常に要求され，以下のような問題点をかかえている．

① 低レベルの割り込み処理中に高レベルの割り込み要求が発生したとき，すみやかに高レベルの割り込み処理へ移行できること．
② 高レベルの割り込み処理終了後，中断された低レベルの割り込み処理があったかどうかを判別し，中断された処理があれば再開できること．

第1の問題に関しては68000にその能力があるので心配ないが，第2の問題を解決するのは容易でなく，このあたりの処理プログラムに問題があると，同じプロセッサを応用した他社のマシンに水をあけられることになる．

3.2 多重割り込みと割り込みレベル

高位レベルの割り込みは68000が起動してくれるが，そのために中断された低位レベルの割り込みは，プログラムで再起動しなければならない．

あるレベルの割り込み処理の終了時には，自分より低レベルの割り込み処理が中断されていないかどうかをチェックする必要があり，しかもそのレベルを順にレベル1になるまでチェックしなければならない．さらには，このチェック中にも，高位レベルの割り込み処理が起動される可能性もあり得る．

このように，いつ発生するかわからない割り込みをソフトウェアで処理するには限界があり，それゆえ割り込みのレベルは，マスカブル割り込みとノン・マスカブル割り込みの2レベル程度でよく，7レベルの割り込みは意味のないものであることがわかる．

以上から，マスカブル割り込み（レベル1～6）のすべてを使うのではなく，レベルをある程度限定し，1つのレベル上に複数の割り込み要求ラインを結線するのが妥当であると思われる．

3.3 多重割り込みを支援するためのハードウェア

優先順位付き割り込み処理を行う場合に，ソフトウェアのオーバヘッドを最小限に抑制するためのハードウェアとして，次の2通りが現実的である．

① 8259を使う．

8259は割り込みコントロール・ユニットと呼ばれるLSIで，途中でこれを操作するので多少のオーバーヘッドは避けられないが，コスト的にも十分一般的である．

② Z-85××シリーズの周辺LSIを使う．

ザイログ社で開発されたもので，シリアル関係のSCC，カウンタ／タイマ／パラレル・インターフェース機能をもつCIOなどがあり，両者とも最高水準のLSIと断言できる性能を有し，割り込みに関するすべての機能が付加されているので，多重割り込みに必要な複雑なソフト的操作は不要になる(割り込み機能だけでなく，LSIそのものが最高水準の機能を有している)．

要点

割り込み処理プログラムを記述するには，68000に関する知識は当然であるが，何といっても，割り込みを要求するディバイスについての的確な知識が要求され，結局ハードウェアに精通することが先決となる．

割り込みを要求するディバイスに関しては，

■そのディバイスが割り込み要求ラインをアサートするには，どのような条件が必要か．

■そのディバイスが割り込み要求ラインをネゲートするのはいつか，あるいは，どう操作すればネゲートできるか．

68000自身に関しては，

■割り込み要求が受けつけられる様子，特に割り込みレベルの設定．

■処理ルーチンはどのように決定されるのか．

■どのようにプログラムを終了したらよいか（RTE命令など）．

割り込み処理ルーチンは，使用するベクタ番号に対応したベクタ領域に用意すればよいが，後はＳＲのＳ，Ｔ，I_2～I_0が操作されることと，スーパバイザ・スタックへはＰＣとＳＲがプッシュされ，割り込みルーチンへ分岐することである．

ＳＲのI_2～I_0を変更する命令は，マスカブル割り込みの許可／禁止を行うので，割り込みレベルの制御には，こらの命令を効果的に使用しなければならない．

以上のことを理解し，１つのディバイスから発生する割り込み処理プログラムを正しく記述し，次に多重処理を目標とすればよい．

要因

外部ディバイスから発生する割り込み要求で，様々な要因が考えられる．

68000の応答

1．割り込み例外処理が開始されるまで

① 割り込み要求ライン上の割り込み要求の様子は，割り込みレベルとしてコード化され，$\overline{IPL_2}$～$\overline{IPL_0}$に与えられる．一方ＳＲの３ビット（I_2，I_1，I_0）には，プロセッサが割り込み処理を受けつけるか否かを判断するレベルを設定しておく．

② コード化された割り込み要求レベルが０（ゼロ）なら，68000は割り込み要求がないものと判断するが，それ以外のコード化された値（０以外）を外部ディバイスからの割り込み要求の発生と解釈する．

③ 68000は割り込み要求を受け取っても即座には割り込み例外処理を実行せず，その処理実行は待機状態となる．待機された割り込みは各命令実行の間に検出され，現在のプロセッサのレベル以下であるなら，割り込み例外処理は起動されない（同レベルでも起動されない）．待機された割り込み（このレベルは少なくともゼロではない）が受けつけられるためには，I_2～I_0に保持されるマスクが変更され，このレベルより高位の要求レベルにならなければならない．ただしレベル７の割り込み要求は，68000の保持するレベルに依存せず常に受けつけられる．

2. 割り込み例外処理の開始

待機させられた割り込みレベルが現在のプロセッサのレベルより高ければ，例外処理が開始される．

① ＳＲのコピーを作成し，この内容を一時的に内部へ退避する．
② ＳＲのＳビットをアサート（スーパバイザ状態），Ｔビットをネゲート（トレース処理を禁止）する．
③ プロセッサの割り込みレベルを，現在受けつけようとしている割り込みレベルにセットする．つまり，ＳＲのI_2～I_0ビットが変更され，このレベル以下の割り込み要求をマスクする．
④ 割り込みアクノリッジ・サイクルへ入り，割り込み要求をしているディバイスからの割り込みベクタ番号を取り込む（ここでは割り込みアクノリッジにはふれない）．このとき，外部ディバイスがベクタの送り出しができず，バスエラーを示している場合，68000はスプリアスとみなし，代わりにスプリアス割り込みベクタ（ベクタ番号24）を内部で発生する．
外部ディバイスからの割り込みベクタのフォーマットは<NOTE　1>，オート・ベクタ割り込みに関しては<NOTE　2>に整理したので，そちらを参照してほしい．

以後は通常の例外処理過程である．

⑤ ＰＣ，ＳＲの順にスーパバイザ・スタックへ退避（PUSH）するが，退避されるＰＣの内容は，割り込みが発生しなかった場合に実行されたであろう命令が記憶されているアドレスである．
⑥ すでに決定されているベクタアドレスの内容をＰＣへ取り込み，割り込み処理ルーチンへ分岐する．

注意事項

■ハードウェア上の問題であるが，外部からの割り込み要求がないときには，$\overline{IPL_2}$～$\overline{IPL_0}$の各端子がすべてHighであるようなロジックとする必要がある．
■レベル７の割り込みはI_2～I_0を操作してもマスクできないので，割り込み要求レベルの設定環境に依存しない割り込みである．
■レベル６～１までのマスカブル割り込みでは，レベルを下げることで，優先度の低い割り込み処理を起動できる．

NOTE

<NOTE　1>　周辺ディバイスからの割り込みベクタ・フォーマット
割り込みを要求した周辺ディバイスは，割り込みアクノリッジ・サイクル中に，８ビットのベクタ番号をデータバスのＤ７～Ｄ０上に送り出す．
データバスＤ15～Ｄ８上の内容は無視され，Ｖ７～Ｖ０で指定できるベクタ番号は64～255であり，これ以外のベクタ番号を使用すべきではない．

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
無					視			Ｖ７	Ｖ６	Ｖ５	Ｖ４	Ｖ３	Ｖ２	Ｖ１	Ｖ０

<NOTE　2>　オートベクタ割り込みについて
割り込みベクタを送り出す機能のないディバイスに対して，オート・ベクタという方法があり，割り込みアクノリッジ・サイクル中に$\overline{VPA}$という端子をアサートすると，その応答として68000は$\overline{VMA}$という端子をアサートし，８ビットマイコンではなつかしい6800のリードサイクルを実行する．そして内部的には，受けつけようとしている割り込みレベルに応じたベクタ番号を自ら発生する．すなわち，この時に内部で発生されるベクタ番号は，25（レベル１）～30（レベル６），31（レベル７）である．

5 アンイニシャライズド割り込み例外処理

●ベクタ番号：〈15〉　●優先度：［Group　1］

概　要

割り込みベクタを送り出せる能力を備えたディバイスが，ソフト的に初期化されていないために，あるいは初期化を忘れてしまって，有効なベクタ番号を送り出せない場合，ベクタ番号15を送り出して一時しのぎをするのがアンイニシャライズド割り込みである．

周辺LSIの初期化が不可能ということはあり得ないので，モトローラではこれから開発を予定している周辺LSIに，この機能を想定しているのではないか，と考えられる．つまり，ハードウェア・リセットによって周辺LSI内のベクタ・レジスタを15（＄0F）に初期化するのであれば，それなりの意味もあるのではないか，と．

たとえば68000ファミリ周辺LSIのMC68230（PI/T）では，割り込みベクタ・レジスタの内容がこのように構成されている．

要　点

ベクタを送り出す周辺LSIが初期化されていないことに起因するので，このことを考慮して処理プログラムを記述する．言うまでもなく，このようなエラーが本番中に発生するようではいけない．

68000の応答

本例外は割り込み例外処理の１つの過程であるから，内容も割り込み例外処理と同様である．

6 スプリアス割り込み例外処理

●ベクタ番号：〈24〉　　●優先度：［Group 1］

概　要

割り込みアクノリッジ中には，割り込み要求状態にある周辺LSIは，$\overline{\mathrm{DTACK}}$か$\overline{\mathrm{VPA}}$のアサートで応答する約束があり，前者はベクタの受け取り，後者はオートベクタであることを68000に知らせるものである．これらのいずれでもない場合には，割り込みアクノリッジサイクルを終了するために，バスエラー・ラインをアサートしなければならない（このようなハードロジックが必要である）．

このような場合，68000はベクタの取り込みに失敗したものと判断し，スプリアスベクタと呼ばれるベクタ番号24を内部で発生させ，スプリアス割り込み例外処理を開始する．

要　点

本例外処理プログラムは，割り込み例外処理中にベクタ番号が得られなかった場合に対処し，割り込み処理の異常からシステムを保護することだが，このような例外処理が起動されたならば，ハードウェアを再検討しなければならない．つまり，ハードウェアの異常そのものをトラップできる点に注目していただきたい．

ハードウェアの異常は致命的であり，本割り込み例外からの回復は困難であるから，この点を十分考慮しなければならない．

68000の応答

本例外は割り込み例外処理の１つの過程であるから，内容も割り込み例外処理と同様である．

7 トレース例外処理

●ベクタ番号：〈9〉　　●優先度：［Group 1］

概　要

トレース例外処理は，〝プログラムをトレースするプログラム〟を開発する際に非常に有効なものである．

開発段階にあるプログラムのデバッグには，アセンブラの命令を１ステップだけ実行し，その結果をチェックする〝トレース〟と呼ばれる機能が不可欠であるが，プログラムをトレースするには，そのように動作するプログラムが必要であり，デバッガと呼ばれるユーティリティプログラムに，このような機能がサポートされていることが多い．

68000は命令実行時にＳＲのＴビットがアサート（ＯＮ）されていると，通常のように次々と命令を実行するのではなく，その命令を実行後トレース例外処理を開始するので，その都度デバッガへ制御を移行し，プログラムの実行状態をモニタできるようなサービスプログラムを実行させる．

要　点

Ｔビットのアサートはスーパバイザ状態でなければ変更できないので，デバッガ自身はスーパバイザ状態でのユーザプログラムを支援するものとなる．

実行させるユーザプログラムからシステムを保護することは当然であるので，スーパバイザ状態～ユーザ状態の間を適切にスイッチすること，システムスタック空間の管理などに注意が必要である．

トレース例外処理シーケンス自身についての理解には，大きな問題はないと思われる．

要　因

Ｔビットがアサートされているときに，命令を１ステップ実行した直後に発生．

68000の応答

① 最初にＳＲの内容を内部にコピーする．
② スーパバイザ状態，トレース禁止状態とする．
　以後例外処理の妨げになるのでトレースは不可能となるが，トレース例外処理プログラム自体をトレースしても無意味であるから，この方が適切なわけである．
③ 内部でベクタ番号を発生し，これを４倍してベクタアドレスを得る．
④ ＰＣ，コピーしておいたＳＲをスーパバイザスタックへ退避するが，ＰＣの値は本例外処理を発生させた次の命令が格納されているアドレスを保持している．
⑤ 所定のサービスプログラムへ制御を移行する．

注意事項

以下の内容はあくまでも一般論であり，各例外処理が起動され，そのサービス・ルーチン内でどのような環境変化があるか，ということによって状況がダイナミックに変化するので，この点にも注目する必要がある．

１：　Ｔビットがアサートされている状態で命令を実行しているときに，ハードウェア割り込みが発生した場合，トレース例外処理が優先され，次に割り込み例外処理が起動される．

2：　命令実行中にその命令自身によって例外処理が発生した場合，発生した例外処理が起動され，その後でトレース例外処理が起動される．
たとえばトレース可能状態でTRAP命令を実行しているときにハードウェア割り込みが発生した場合，以下の順に処理される．

①　TRAP命令を実行
　　トレースも割り込みも実行中の命令を中断させるものではなく，実行中の命令が終了してから例外処理を開始する．
②　トレース例外処理を実行
③　割り込み例外処理を実行

3：　トレース例外処理が発生しないケース
■割り込み例外処理が開始されたために命令が実行されなかった場合．
■実行すべき命令が不当命令，特権命令であった場合．
■ハードウェアリセット，バスエラー，アドレスエラーによって命令が中断された場合．

8 TRAP命令例外処理

●ベクタ番号：〈32〉〜〈47〉　●優先度：［Group 2］

概　要

システム・プログラムの設計者は，〝プログラムの階層化〟ということを常にイメージしており，ユーザ・プログラムのTRAP命令によって基本機能のサービスが行えるように，システムを設計する．

TRAP命令はユーザ状態でスーパバイザへのサービスを要求するもので，（スーパバイザコールまたはシステムコールという）要するに基本サブルーチンの呼び出しに使う命令である，と解釈してもよい．

TRAP命令のオブジェクト・フォーマットの関係から，命令で指定するベクタ番号０〜15が実際の例外ベクタ番号32〜47に対応する．またTRAP命令によるサービス情報は，その時のデータレジスタやアドレスレジスタに格納されてスーパバイザ・プログラムに渡されることが多い．

要　点

サービス・プログラムはサブルーチンの作成と同様であり，入／出力条件を無駄なく構成しなければならない．

処理内容としては，渡された引数を解析し，それに応じた処理を実行し，結果を所定の場所へ格納（時にはエラー情報を返す）し，RTE命令でもどればよい．

要　因

TRAP命令自身の実行による．

68000の応答

① ＳＲを内部にコピーし，スーパバイザ状態，トレース状態をオフとする．
② TRAP命令で指定されているベクタ番号に対応し，32〜47までのいずれかのベクタ番号を内部で発生する．その後発生したベクタ番号からベクタ・アドレスを得る．
③ ＰＣをスーパバイザ・スタックへ退避するが，退避されるＰＣの内容は，例外処理を発生させた次の命令が格納されているアドレスを保持している．
次にコピーしたＳＲをスーパバイザ・スタックへ退避する．
④ 例外ベクタを参照し，サービス・プログラムへ制御を移行する．

APPENDIX●プログラムの階層化とTRAP命令

プログラムを開発する際には，プログラムの階層化とか基本サブルーチンという概念は非常に重要なので，簡単にふれておきましょう．

68000を搭載した1枚のボードが,どのように機能するかはそのソフトウェアに依存するわけだが，応用分野が異なるたびに新規に作成する必要がない部分もある．

たとえばそのボードに対するデータの入／出力形式は同一だが，ボード内の処理系のみ異なる，というのであれば，入／出力プログラム（基本サブルーチンともいう）の呼び出し形式を統一することで，処理系だけの開発に専念できるのである．

このように，ある階層に属するプログラムを統一的に呼び出せるようにTRAP命令が用意されており，先の例では共通のプログラムの呼び出しにTRAPが有効であるとしたが，OSの基本入／出力ではこの様子がさらに徹底されている．

たとえば文字表示に必要なプログラムは各コンピュータ間で異なるが，文字表示のための手続きは統一されているので，プログラマの記述する文字表示プログラムは同じになる．

つまり入り口までの手続きを統一しておけば，その先で実行されるであろう文字表示のためのプログラム（CRTコントローラの操作など）の相違は，プログラマには関係のないことになる．

9 Zero Divide CHK TRAPV などの命令による例外処理

●ベクタ番号:〈5〉
●ベクタ番号:〈6〉
●ベクタ番号:〈7〉

●優先度:［Group　2］

要　点

いずれもトラップ発生用命令に分類され，ユーザプログラム内からトラップを発生できるが，これらは条件付きでトラップを発生し，TRAP命令のように必ず発生するのではない．その他の詳細はTRAP命令による例外処理と同様である．

表1.33　トラップ発生用命令の分類

Zero Divide	除数0（ゼロ）で除算した場合に例外処理が起動されるが，オーバフロー発生時には例外処理が発生しないので，必要とあらば除算命令の直後にTRAPV命令を置いて対処する． 内部で発生されるベクタ番号は5である．
CHK	メモリ空間の境界を管理するために用いられ，オペランドに設定した内容からメモリ境界の侵犯をチェックでき，レジスタ値が0（ゼロ）より小さいか，ソース・オペランドの上限値より大きいと例外処理が起動される． 内部で発生されるベクタ番号は6である．
TRAPV	前の命令を実行し，その結果CCRのVビットがセット（〝1〟）である場合にトラップ例外処理が起動されるので，オーバフローが発生し，そのために特定のプログラムへ制御を移行したい場合，そのような命令の直後にTRAPVを置く． 内部で発生されるベクタ番号は7である．

68000の応答

TRAP命令による例外処理と同様

10 不当命令 ●ベクタ番号：<4>
未実装命令 ●ベクタ番号：<10，11>
の実行による例外処理

●優先度：［Group 1］

概 要

1．未実装命令：ベクタ番号10，11

命令の第1ワードのビット15〜11までのビットパターンが〝1111〟または〝1010〟である命令は存在せず，これらに該当する命令を実行しようとすると，未実装命令例外処理が発生する．このように，命令セット以外の命令を実行させれば特定のプログラムへ制御を移行できるで，ユーザ定義の新しい命令として利用できる．また未実装命令のビットパターンのビット11〜0にも，ユーザ側で〝意味〟をもたせることも可能である．

内部で発生するベクタ番号は，〝1010〟のパターンではベクタ番号10，〝1111〟ではベクタ番号11となっている．

2．不当命令：ベクタ番号4

不当命令例外処理は，命令の第1ワードのビットパターンが正当なパターンではなかった場合に発生する．

不当命令によるオブジェクトコードのビットパターンを例外処理プログラムで解析するのは得策ではなく，活用するのであれば，未実装命令による例外処理の方が混乱しないと思われる．

要 点

不当命令による例外処理はプログラムの開発段階では重要であり，デバッガでは必ずサポートされなければならない機能である．もし本番中にこのようなエラーが発生するなら，プログラムをもう一度検討しなければならない．

不当命令例外を意識的に使用しないという前提なら，デバッガでのサービスでは，不当命令を取り込んだ場所をプログラマに通知してシステムへもどり，本番中では，原因となった命令を取り込んだメモリアドレスをどこかへ記憶し，次の命令から継続するとか，処理の流れを別方向へ進める，などの対策が考えられる．

いずれにしてもこのようなエラーが検出されたならば，プログラムを修正しなければならない．

未実装命令例外処理は基本的にエミュレーションであるから，関数（サブルーチン）と同様な処理を行ってRTEでもどればよい．

68000の応答

TRAP命令時の例外処理シーケンスと同様である．

11 特権違反による例外処理

●ベクタ番号：〈8〉　●優先度：［Group 1］

概　要

システムの信頼性を確保するために幾つかの命令が特権化され，これらの命令をユーザ状態で実行すると本例外が発生する．

内部で発生するベクタ番号は8である．

【特権命令】

STOP
RESET
RTE
MOVE to USP
MOVE to SR
ANDI to SR
ORI to SR
EORI to SR

要　点

特権命令はユーザプログラムでは実行する意味がないわけで，これが検出されることはプログラムエラーである．そこでデバッガでのサポートでは，エラーの原因となったアドレスを表示してシステムへもどり，本番中でのサポートは，エラー発生の原因となったメモリアドレスをメモリのどこかへ記憶し，次の命令から継続するとか，あるいはプログラムの流れを別方向へ移行する，という処理プログラムになる．

いずれにしてもこのようなエラーが検出されたならば，プログラムを修正しなければならない．

要　因

ユーザ状態で特権命令を実行したとき．

68000の応答

TRAP命令時の例外処理シーケンスと同様である．

第2部　プログラム

プログラミング編の構成ですが，プログラミングに必要な知識とその解法を，短期間で無駄なくマスタできるように配慮しました．そこで，まず「問題点」を提起し，それには「どのように対処すればよいのか」，というような流れになっています．

本編は次のような各章から構成されます．

1　ツールとしてのアセンブラ
2　基本サンプルプログラム
3　プログラム制御および汎用サブルーチン

以下は各章の「ねらい」を整理したものです．

1　ツールとしてのアセンブラ

アセンブラでプログラミングする際には，エディタで作成されたソースファイルがアセンブルされ，目的とする機械語に変換された後，さらにデバッガでデバッグすることになるのですが，このような過程は，解説編での内容とは異質の分野であり，アセンブリ言語の仕様とその基本操作など，アセンブリ言語による開発環境の現実やその可能性について言及しています．

また，本書のプログラムをデバッグするために使用した固有の環境についてもふれ，例題を他機種上で走行させる場合のことも考慮しました．

2　基本サンプルプログラム

個別命令の理解のためのセクションで，たとえば，データ転送命令を拡張したブロック転送命令，加算命令を拡張して倍精度演算をするとか，実例を示した方がよいであろうと思われる命令など，個別命令を中心に話を進めています．

3　プログラム制御および汎用サブルーチン

プログラミングの際に知っておくべき基本事項として，サブルーチンの構成，ジャンプテーブル，制御構造，などの解説をしています．

また，使用頻度の高いサブルーチンの作成方法についても，考え方を中心に解説しましたが，実用性にも十分な配慮をしました．

◎ツールとしてのアセンブラ

1 テキスト・エディタ

アセンブラで開発をするには，まず優秀なテキスト・エディタが必要であり，さらにクロスアセンブラを使えば，1台のコンピュータ上で様々なMPU(CPU)のプログラム開発が可能であるばかりか，続々と発表される新機種のコンピュータに心を奪われることもありません．

特に現在市販されている16ビット・パーソナルコンピュータは，開発マシンとしても十分すぎるパフォーマンスを備えていて，特定の場所から供給されるツールではなく，ユーザがツールを選択できる時代になりました．

プログラミングの第1歩は，いうまでもなく，アセンブラのニーモニックを記述したソースファイルの作成であり，この作成にはテキスト・エディタ（単にエディタともいう）と呼ばれるツールが必要です．しかも，ひょっとすると5年や10年は，お世話になる可能性があります．

以下はエディタに要求される主な機能で，最低限度の機能として，この程度はサポートしてほしいと思っているスペックです．

■高速であること

キー入力のスピードに文字表示が追いつかないとか，画面のスクロールが低速では，とても仕事になりません．

また，この部分が高速なエディタは，文字列の検索や置き換えも高速です．

■キーボードのインスツールが考慮されていること

エディタに要求される機能の呼び出しが固定されていては，新しいエディタを購入するたびに，そのエディタに合った使い方を強要されてしまいますが，機能をユーザがカスタマナイズできるエディタも市販されています．

たとえば，1行の削除を実行するにしても，エディタの開発者の趣味に従うのではなく，ユーザが独自にインスツールできるものがあります．

■ほとんどの基本操作がコントロール・キーとの併用で行えること

〝A〞～〝Z〞までをコントロール・キーと同時に押すことで，テキストの編集に必要な，換言すれば，頻繁に使う編集機能が利用できると，テキストの作成は飛躍的に向上します．

一般には，アルファベットを2文字まで許しており，必要な作業のすべてを，このように利用できるエディタがあります．

■マクロ・コマンドがサポートされていること

これは，編集機能を手作業で行うのではなく，どのように編集するかということをプログラムし，1つの新しい編集コマンドとして置き換えることのできる機能です．

■横スクロールをサポートしていること

経験上，132文字以上は1行として必要であり，横方向のスクロールがサポートされねばなりません．

■編集機能

エディタを商品として売るからには，それなりの機能は備えていると考えてよいでしょう．

高価な開発マシーン上では，EMACSという優れたエディタがありますが，パソコン上で，以上のスペックを満足する商品となると，非常に限られているのが現状で，なかには，「いかに簡単に使えるか」と言いたいのでしょうが，新入の女子事務員向けとも受け取れるような広告を技術誌に出すものまであります．

以上の条件をすべて満たす商品が2点あります．

1つはPMATEで，もう1つはFINAL(SPS－Edit)です．

個人的にはPMATEを使用していますが，非常に多機能で，「よくもこれだけの機能を詰め込んだものだ」と感心させられます．宣伝するわけではありませんが，FINALもすばらしく，コストの点からこちらがよいと思います．これらは，これから開発されるであろう様々なマシン上に移植され，末長く愛用されることでしょう．

2 MS-DOS上のクロスアセンブラ

68000の開発は，68000を搭載したコンピュータでなければできないわけではなく，要するにプログラマの記述した，

```
MOVE.W   (A0),D0
```

のようなニーモニックを解釈し，68000の機械語に翻訳（アセンブル）できるような処理プログラムがあればよく，このような処理プログラムが，Z-80上のプログラムであろうと，8086上のものであろうと，あるいは6809上のものであろうと，いっこうにかまわないわけです．

以上のような機械語翻訳処理プログラムを，クロス・アセンブラといい，コンピュータが1台あれば，そのコンピュータ上で走るクロス・アセンブラを用意するだけで，様々なプロセッサの開発を行うことが可能です．クロスCコンパイラでも不可能ではありませんが，言語仕様やコスト面などで，クロス・アセンブラの方が現実的です．

■XA68Kについて

本書ではMS-DOS上の68000クロスアセンブラとして，NECのソフトウエア・リストにある〝XA68K〟というツールを使用しましたが，機能はかなり低く，それだけにプログラマの手作業に依存する部分もあり，仕事ができないわけではないのですが，「これで行こう」と思わせるツールではありません（文法はモトローラ社に準拠し，マクロ機能はサポートされません）．

しかし，商品価値がないわけではなく，機器組み込み用とかモニタ・プログラムなどの小さなプログラム開発，アセンブラの教育用に向いています．

ただし，**図2.1**のようなバグもあり，CCR形式を指定してもSR形式の機械語コードを出力するので，機械語コードを手作業で修正する必要があります．そこで，本文中では修正した正しいリストを掲載しました．

APPENDIX●いろいろなアセンブラ

◉レジデント・アセンブラ

CP/M-68Kなどのように，68000上で68000のアセンブルをするプログラムを単にアセンブラ，あるいはレジデント・アセンブラと呼びます．つまりオブジェクトコードの生成過程で，レジデント・アセンブラとクロス・アセンブラという区別がなされるわけです．

◉アブソリュート・アセンブラとリロケータブル・アセンブラ

ソース・ラインをいきなり機械語へ変換するアセンブラをアブソリュート・アセンブラ，一度リロケータブル・オブジェクト・コードへ変換し，これをリンカへ入力して機械語を生成するアセンブラを，リロケータブル・アセンブラといいます．

これは，レジデント・アセンブラ／クロス・アセンブラのいずれにもあてはまる問題であり，クロス・アセンブラでもリロケータブルなコードを出力できる高級品が市販されています（**18 モジュール別開発と市販ツール**参照）．

アブソリュート・アセンブラ：小規模な開発向け
リロケータブル・アセンブラ：大規模な開発向け

図2.1 サンプルプログラム：BUG

```
LINE ADDR. CODE/VALUE        LABEL    OP      OPERAND        COMMENT

0005
0006        =00000000        FLG      EQU     0
0007
0009        =00005000                 ORG     $5000
0010        =00005000        BUG
0011 005000 027C 0000                 ANDI    #FLG,SR
0012 005004 027C 0000                 ANDI    #FLG,CCR
0013
0014 005008 007C 0000                 ORI     #FLG,SR
0015 00500C 007C 0000                 ORI     #FLG,CCR
0016
0017 005010 0A7C 0000                 EORI    #FLG,SR
0018 005014 0A7C 0000                 EORI    #FLG,CCR
0019
0020        =00005000                 END     BUG
```


◉マクロ・アセンブラ

複数の命令群をユーザが定義した仮想命令で置き換える機能で，いくつかの命令を一括して定義することができます．これらの命令群が文字を出力するのであれば，PRINTという名前でマクロ定義すると，以後，PRINTと記述するだけで定義した命令群の呼び出しが可能になります．もちろん，引数を渡すこともできます．

68000にはPUSH/POPという命令がデータ転送命令に汎用化されているので，PUSH/POPというニーモニックは存在しません．そこで，スタック操作をどうしてもPUSH/POPで記述したければ，スタックとのやりとりをPUSH/POPという文字例でマクロ定義してしまいます．このようにすれば，あたかもPUSHやPOPが68000の命令セットであるかのように扱うことができます．

マクロ機能を使えば，プログラマの趣味に合った記述をすることができ，開発環境は向上します．

3 アセンブラの仕様

プログラミング言語の仕様ですが，8086と68000とは異なるプロセッサであり，ニーモニックが同じでないのは当然です．

68000というプロセッサの開発をアセンブラで行う際に，同じ68000のアセンブラでも，その仕様が異なっていては都合が悪いので，アセンブリ言語の標準は，プロセッサの開発をしたモトローラ社に準拠するのが常識となっています．ただし，言語仕様に関しては，ツールの開発者の趣味に依存するわけですから，その仕様を確かめることも大切です．

XA68Kのアセンブリ言語について，基本的な事柄を整理しますが，言語仕様のすべてを本書に掲載することは不可能であるため，サンプル・リスト中に新しい表現が使用され，そのためにリストが読めないということがあれば（このようなことは，まず発生しないと思われます），そのつど必要な解説をすることにします．

［1］ ソース・ラインの形式

アセンブラへ入力される行は，次のような4つのフィールドに分類することができ，アセンブラはこれらを解釈し，直接機械語へ変換するか，リロケータブル・アセンブラなら，リンカへ入力するためのリロケータブル・オブジェクト・コードへ変換します．

① ラベル・フィールド
② オペレーション・フィールド
③ オペランド・フィールド
④ コメント・フィールド

4つのフィールドは1個以上のスペース・コードまたはタブ・コードで分離され，コメント・フィールド以外の3つのフィールドは，スペース・コードまたはタブ・コードによって，フィールドの終わりとみなされます．

■ラベル・フィールド

ラベルはジャンプ先アドレスやデータ領域のアドレスを記号（シンボル）で表現するもので，アセンブラを使う大きなメリットでもあり，ラベル（シンボル）の使えないアセンブラは存在しません．

ラベル・フィールドはソース・ライン中の最初のフィールドに位置し，第1文字目から記述しますが，ラベル・フィールドの存在しない行の第1文字は，スペース・コードかタブ・コードでなければなりません．また，ラベルだけの行も許されます．

■オペレーション・フィールド

命令のニーモニックが記述されるフィールドです．

■オペランド・フィールド

命令のオペランドが記述されるフィールドです．

■コメント・フィールド

セミコロン（；）から行の終わりまでがコメント・フィールドであり，任意の文字例を

記述することができます．本フィールドは，オブジェクトの生成には無関係なフィールドです．

■行全体がコメント行と解釈される行

●行第１文字がアスタリスク（*）である行．

●スペース・コードかタブ・コードしか存在しない行．

［2］ 識別名

識別名には，ユーザが定義するシンボル，アセンブラに対してすでに意味の定まっているキー・ワード，の２つがあります．

■シンボル

シンボルの属性は**表2.1**のようになりますが，ラベル，数値，変数は，明確な区別が要求されるわけではありません．

表2.1　シンボルの属性

ラベル	命令のロケーション・アドレスの値を保持する． 一般に分岐先の指定などに使用される．
定　　数	永久割り当てが行われた値を意味する． EQU疑似命令によって指定したシンボルに値が定義され，以後，このシンボルを定数として使用できる． 途中で値の変更をすると２重定義となり，アセンブルエラーが発生する．
変　　数	一時的に割り当てが行われた値を意味する． ここでの変数とは，メモリ上に確保したエリアとしての変数ではなく，SET疑似命令によって値が定義されるもので，EQUで定義されたシンボルと異なり，途中でその値を変更できる．
レジスタ・リスト	REG疑似命令で定義されるもので，MOVEM命令のオペランドで使用されるものである．
アスタリスク(*)	特別な変数であり，現在のロケーション・アドレスを保持している．

■キー・ワード

命令のニーモニックやレジスタ名などがあります．

■識別名の定義

①　最初の１文字は英文字（大文字または小文字）かアンダーライン（＿）である．

②　２文字目以降は，上の条件に英数字を加えた文字が許される．

③　最初の８文字を有効とする．

［3］ 定数

定数には，数値定数，文字定数，文字列定数（ストリング）があります．

■数値定数

％（２進），＠（８進），＄（16進），10進（識別子なし）の４つの表記が可能で，サイズは

32ビットとして扱われます．また，少ない桁数が指定されると，アセンブラが上位にゼロを満たします．

［例］　%1010
　　　　@210
　　　　＄FD01(小文字も許され,$fd01と記述してもよい)
　　　　256

■文字定数

シングルクォーツ（’）で囲まれた文字を文字コードとして扱います．

［例］　’F’　(16進では＄41)

■文字列

ダブルクォーツ（”）で囲まれた部分が文字列です．

［例］　〝String〞

［4］ 演算子

算術演算，論理演算，比較演算，がサポートされます．

■算術演算子

演算は符号付き整数として扱われます．

－n　　：nの2の補数を式の値とする．
n＋m　：nとmとの和を式の値とする．
n－m　：nとmとの差を式の値とする．
n＊m　：nとmとの積を式の値とする．
n／m　：nとmとの商を式の値とする．
n％m　：nとmとの余りを式の値とする．

■論理演算子

n＆m　：nとmとのビットごとの論理積（AND）を式の値とする．
n！m　：nとmとのビットごとの論理和（OR）を式の値とする．
n＾m　：nとmとのビットごとの排他的論理和（EOR）を式の値とする．
n＞＞m：nをmビットだけ右へシフトする．(最上位ビットは変化しない)
n＜＜m：nをmビットだけ左へシフトする．(最下位ビットには0が入る)

■比較演算子

比較演算子は，2つの符号付き整数を比較し，結果が真ならー1（マイナス1，$FFFFFF-FF）を，偽ならゼロ（0）を式の値とします．

n＝m　：nとmが等しいときに真．
n＜＞m：nとmが等しくないときに真．
n＞＝m：nがm以上のときに真．(n＝＞mでも同じ)

n＞m　：nがmより大きいときに真.
n＜=m：nがm以下のときに真.（n=＜mでも同じ）
n＜m　：nがmより小さいときに真.

［5］ アセンブラ疑似命令(ディレクティブ)

疑似命令とは何か，どのような働きをするか，ですが，これには，アセンブラというマシン語変換プログラムが，どのような「処理」をするのか，ということを考えてみる必要があります．アセンブラというプログラムは，プログラマの作成したソース・ファイルを読み込み，記述されているニーモニックを該当するマシン語へ変換する作業を行いますが，この過程では，アセンブラ自身がプログラマからの指示を仰ぐ部分があります．

たとえば，単にMOVEというニーモニックをマシン語へ変換する以外に，アセンブラへの制御を指示する命令も必要であり，この命令はマシン語へ変換される情報とは区別されねばならず，この種の命令を疑似命令と呼びます．

疑似命令にも様々なものがあり，慣れないうちはよく理解できない面もあろうかと思いますが，アセンブラのマニュアルに記述されているすべての疑似命令を理解する必要もなく，本書で解説される基本的なものを理解すれば，かなりの部分までカバーできるはずです（アセンブラの疑似命令の考え方は，そのままCコンパイラのプリプロセッサに直接反映されています）．

■アセンブル時の制御に関するもの

ORG　；書式　ORG　＜式＞

① ロケーション・アドレスを式の値にセットするもので，アセンブラはこの値をたよりにアドレスの割り付けを行う．
② 式にシンボルを記述する場合には，すでに定義されていなければならない．
③ 高級なリロケータブル・アセンブラでは不要である．

［例］　ORG　$5000

END　；書式　END　＜式＞

① これ以降には有効な行がないことをアセンブラへ知らせるが，ENDがなくても，ソースファイルの終わりに到着すれば，その時点でアセンブルを終了する．
② ＜式＞があれば，式の値をプログラムのエントリ・アドレスと解釈する．

［例］　END　ENTRY

■シンボルの定義に関するもの

EQU　；書式　＜シンボル名＞　EQU　＜式＞

① シンボルの値を式の値にセットし，以後，このシンボル名を定数として使用できる．
② 式にもシンボルを記述できるが，このシンボルはすでに定義されたものである必要がある（前方参照できない）．
③ 同じシンボル名を再定義できない．
④ ソースファイルの先頭部分に一括して定義した方がよく，命令ニーモニックの途中

にゴテゴテ記述すべきではない．

［例］　DATA　EQU　$56F0

SET　；書式　<シンボル名>　SET　<式>

① シンボルの値を式の値にセットし，以後，このシンボル名を定数として使用できる．
② 式にもシンボルを記述できるが，このシンボルはすでに定義されていなければならない（前方参照できない）．
③ 同じシンボル名を再定義しながら使用できるので，変数に分類される．

［例］　DATA　SET　$2000

REG　；書式　<シンボル名>　REG　<レジスタ群>

MOVEM命令で指定するレジスタ群を，定義したシンボル名で置き換えるためのもので，定義方法や呼び出しの方法がアセンブラによって異なる（本書では混乱を考え使用しないが，このような疑似命令がある）．

■データ定義／メモリの割り付けに関するもの

DC　；書式　<ラベル>　DC{.B.W.L}　<データ列>

① データ列フィールドで指定されたデータを，現在のロケーション・カウンタで示されるアドレスへ割り当てるが，1個のデータを定義してもよく，〝データ列〟にこだわる必要はない．
② {.B.W.L} はサイズを意味するが，省略はワード（.W）と解釈する．
③ 複数データをカンマ（,）で区切って同じ行に記述してもよい．
④ どこへ割り付けるかはプログラマの自由であるが，通常のニーモニックが記述される空間とは別に管理すべきである．

［例］
```
MSG      DC.B  "Message"
DATA     DC.B  $FF,$20,$6F
L__DATA  DC.L  -1,XYZ
```

〝Message〟の〝M〟が格納されるアドレス（先頭アドレス）は，アセンブラによって，MSGというラベルに割り付けられる．その他のラベルも同様に，〝アドレス値〟がアセンブラによって算出される．

DS　；書式　<ラベル>　DS{.B.W.L}　<式>

① 式で指定されただけの領域を確保する．
② {.B.W.L} はサイズを意味するが，省略はワード（.W）と解釈する．
③ どこへ確保するかはプログラマの自由であるが，通常のニーモニックが記述される場所とは別の空間をデータ領域として管理した方がよい．

［例］
```
BUFFER  DS.L  256   ；ロングワードデータを格納する場所を256個確保する．
WORK    DS.B  LEN   ；バイトデータを確保する場所をLEN個確保する．
```

BUFFERやWORKは，確保された先頭アドレスを保持するために，アセンブラが適切なアドレス値を生成する．

4 例題の走行環境

本書の例題は筆者の所有する68000ボード上で実際に走らせましたが，本ボード上では比較的強力なモニタ機能を利用できます．ここではモニタでサポートされているCP/Mライクなファンクション・コールについてふれておきます．

ファンクション・コールの手続きは以下のように，Ｄ０の下位バイトにファンクション・コードを入れてTRAP　#0を実行しますが，リターン情報に使われるレジスタ以外のすべてのレジスタは，ファンクション・コールによって破壊されません．

```
MOVE.B #<ファンクション・コード>, D0
TRAP   #0
```

表2.2　ファンクションコールの手続き

＄００：　プログラムの終了	MOVE.B #0,D0 TRAP　#0
＄０１：　キーボードから１字入力	MOVE.B #1,D0 TRAP　#0
キーボードから１文字入力されるまで待ち，入力されたキャラクタをＤ０の下位バイトへ格納してもどる．オペレータの入力したキャラクタはスクリーンへエコー・バックされる（CTRL＋Cのチエックが行われる）．	
＄０２：　スクリーンへ１字出力	Ｄ１．Ｂへ文字コードをセット MOVE.B #2,D0 TRAP　#0
Ｄ１の下位バイトの内容をスクリーンへ出力する．	
＄０３：　補助入力	MOVE.B #3,D0 TRAP　#0
補助入力装置から１文字入力されるまで待ち，入力されたキャラクタをＤ０の下位バイトへセットしてもどる．	
＄０４：　補助出力	Ｄ１．Ｂへ文字コードをセット MOVE.B #4,D0 TRAP　#0
Ｄ１の下位バイトの内容を補助出力装置へ出力する．	
＄０５：　プリンタ出力	Ｄ１．Ｂへ文字コードをセット MOVE.B #5,D0 TRAP　#0
Ｄ１の下位バイトの内容をプリンタへ出力する．	

次ページへ続く

＄０６：　直接コンソール入／出力

```
MOVE.B  #$FF,D1     ；入力
MOVE.B  #6,D0
TRAP    #0
```

Ｄ０の下位バイトが＄ＦＦならキーボード入力を行う．
キーボードの準備ができていればＤ０の下位バイトに入力した文字コードをセットしてもどるが，入力がなかった場合はＤ０の下位バイトに＄００がもどされる．

```
MOVE.B  #<コード>，D1   ；出力
MOVE.B  #6,D0
TRAP    #0
```

Ｄ１の下位バイトが$FF以外なら，Ｄ１には文字コードがセットされているものと解釈し，それをスクリーンへ出力する．

＄０７：　直接コンソール入力

```
MOVE.B  #7,D0
TRAP    #0
```

キーボードから１文字入力されるまで待ち，入力された文字コードをＤ０の下位バイトへセットして戻る．

＄０８：　エコーなしのキーボード入力

```
MOVE.B  #8,D0
TRAP    #0
```

オペレータのヒットしたキーがスクリーンへ表示されない以外は機能＄０１と同様である．

＄０９：　文字列のスクリーン出力

Ａ０に文字列の先頭アドレスをセット

```
MOVE.B  #9,D0
TRAP    #0
```

｀＄′コードで終了する文字列の先頭アドレスをＡ０へセットしてTRAPを実行するが，｀＄′コードそのものは表示されない．

＄０Ａ：　バッファード・キーボード入力

Ｄ１．Ｂにバッファの長さをセット
Ａ０にバッファの先頭アドレスをセット

```
MOVE.B  #$A,D0
TRAP    #0
```

１行の入力を行うもので，C/R（キャリッジ・リターン・コード）が入力されると作業を終了する．
Ａ０に文字バッファの先頭アドレス，Ｄ１の下位バイトには＜１行の最大値＋１＞をセットするが，〝＋１〞はC/Rコード用のスペースとして必要である．
本ファンクション・コールにより，バッファの先頭にはバッファの長さ（Ｄ１で指定したもの），次には入力された有効文字数（C/Rを除く），３番目のアドレス以降にキーボードから入力された文字が順に格納され，入力された文字列の最後にはC/Rが付加される．
もしC/Rが入力されず予定の入力をオーバーするようなことがあっても，指定文字数に達するとそれ以上の入力を受けつけないようになっている．

＄０Ｂ：　キーボード・ステータスの検査

```
MOVE.B  #$B,D0
TRAP    #0
```

キーボードからの文字入力が可能（タイプ・アヘッド・バッファに文字が取り込まれている）であればＤ０の下位バイトに＄ＦＦが，そうでなければ＄00が返される．

＄０Ｃ：　バッファを空にしてからのキーボード入力

D1.Bに機能コード

```
MOVE.B  #$C,D0
TRAP    #0
```

はじめにキーボード・バッファをリセットし，次にＤ１の下位バイトにセットしたファンクション（＄01，＄06，＄07，＄08，＄０Ａ）を実行する．

その他ファイル・アクセスや68000のベクタ領域の参照と設定，スーパバイザ・モードへの移行，メモリ状態の参照と設定，などが利用できますが省略します．

5 データ転送命令サンプルプログラム

1 連続したメモリ領域をゼロで満たす

◉サンプルプログラム ［FILL］

ある長さをもったメモリ領域（これをブロックと呼ぶ）をゼロで満たすもので，ファイルバッファを初期化するとか，スクリーンをクリアするとか，あるいは，プログラムのデバッグにも有効であり，たとえば，まずゼロで初期化しておき，作成したプログラムを走らせた後でこのエリアの変化をチェックしてみるなど，様々な使われ方をします．

■動作

MEM_STという場所から，ロングワード（4バイト）を単位として10回ゼロを書き込むもので，MEM_STからの40バイトがゼロで満たされる．

■要点

① 「連続した」という条件によって，メモリアドレスを直接指定するアドレッシングではなく，(An)＋のアドレッシングを使ってメモリへ書き込むのが妥当である．

② メモリをゼロで満たすには，CLR（クリア命令）も使用できるが，連続したメモリ領域への書き込みには，データレジスタの内容をメモリへ転送するのが一般的である．

③ ある動作（ここでは，メモリへの書き込み）を指定回数実行するには，DBRAという命令を使い，回数をデータレジスタで指示する．

■レジスタの割り当て

A0：ゼロを満たすべきメモリアドレスを保持する．

D0：繰り返しの回数を指定するが，10回なら1少ない9をセットする．

D1：この内容をメモリへ書き込むが，終始ゼロを保持することにする．

■各行の意味

行6：ORGにより，本プログラムのオブジェクトが＄5000から配置されることをアセンブラへ知らせる．
高級なリロケータブル・アセンブラでは絶対番地を指定せず，アセンブラの出力するオブジェクトをリンカへ入力し，実行可能なオブジェクトはリンカから得られる．

行8：アスタリスク（＊）はコメントであり，プログラムの保守や第3者への配慮のために使用し，アセンブラはソースラインの先頭が`＊´であると，以後はアセンブルに無関係であると解釈する．
14行のオペランドの後方にセミコロン（；）があるが，やはり，以後にはコメントが位置していることをアセンブラに知らせるためのものである．
ただし，オペランド部の次にスペースまたはタブを置けば、それだけでオペランド部が以後に存在しないと解釈するアセンブラが一般的であるので，セミコロンが不要であるアセンブラの方が多い．

行9：何も記述されていないが，このような行も時には必要であり，リストを読みやすくする．

行14：MEM_STはアドレスを保持し，ここでは＄5014というアドレス値をアセンブラがアセンブル時に求める．プログラマはシンボルを介してアドレスを得ることができ，A 0は＄5014に初期化され，このアドレスからゼロを書き込んで行く．

行15：18行の命令を何回繰り返すかを指定するループカウンタの初期化をしているが，ここでは10回実行することにした．
#10－1 の部分は，1つ少ない値をループカウントとすべきことを明示するもので，9と記述してもよい．
このように，オペランドには式を記述することも許される．

行16：D 1を初期化する行で，この値がメモリへ転送されることになる．

行17：CLR_LOOPは分岐先のシンボルで，以降には何も記述されていないが，このようにするとプログラムが見やすくなる．つまり，レジスタの初期化部とクリア部とを明確にしているわけだ．

行18：実際にメモリをクリアする命令が位置している．
命令実行後，サイズはロングワードなので，A 0は勝手に4だけ増加する．

行19：D0の内容が$FFFF でなければ，CLR_LOOPというアドレスへ分岐し，$FFFF（－1）になると次の行へ制御を移行する．

行29：DSはメモリ領域を予約する疑似命令で，ソースラインの位置は命令ニーモニックの所へ記述する．
オペランドは 10＊4 であり，式の値（40）だけロングワードの領域を確保するから，全体では160バイト（40＊4）だけのメモリ領域を予約している．

DS.B 10＊4　なら，40バイト
DS.W 10＊4　なら，80バイト（40＊2）

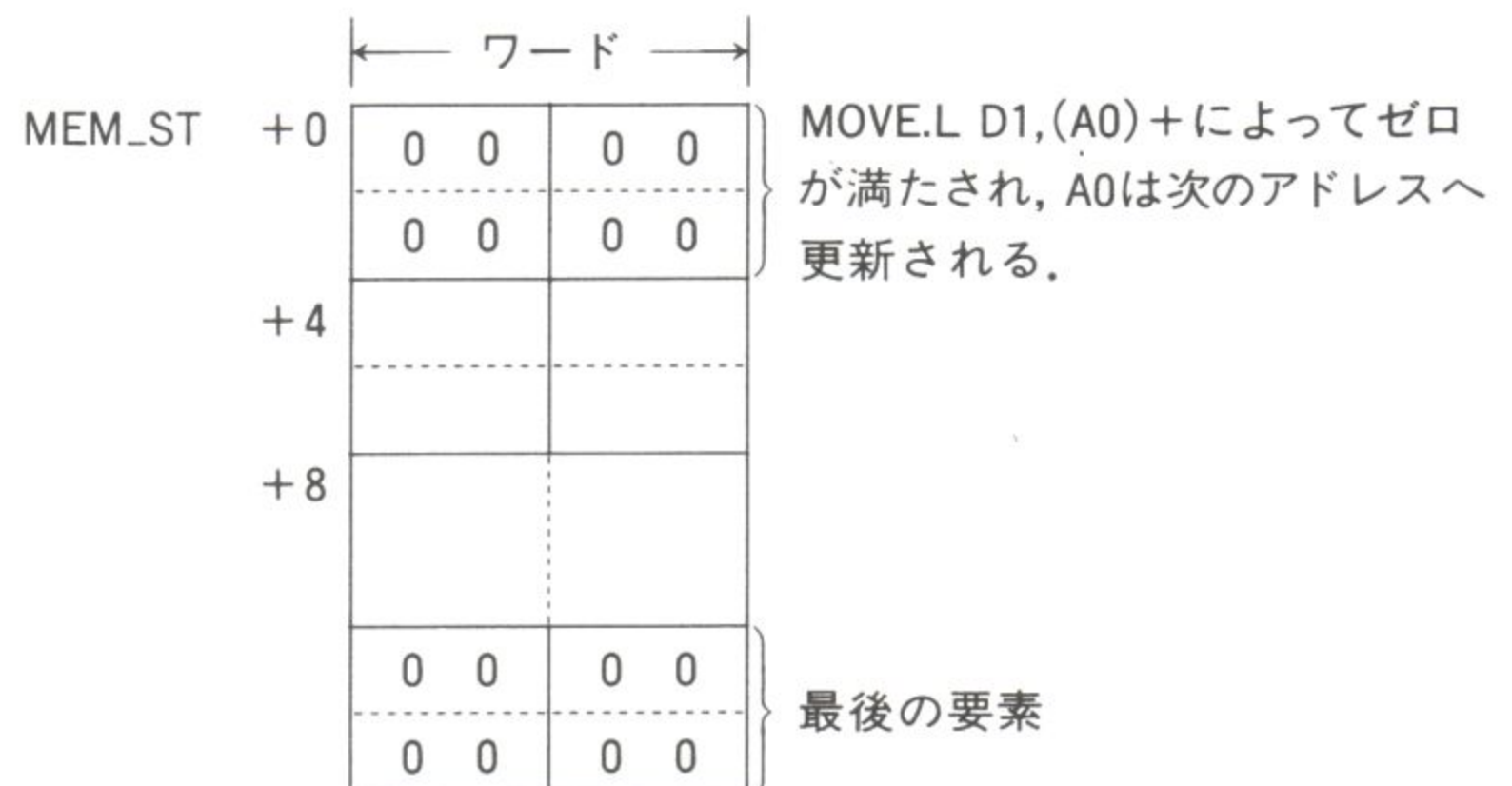
のメモリ領域を予約することになる．

リスト［FILL］

```
LINE ADDR.  CODE/VALUE          LABEL     OP        OPERAND         COMMENT

0005
0006        =00005000                     ORG       $5000
0007        =00005000
0008                            *                  memory clear
0009
0010                            *                  A0 :start address
0011                            *                  D0 :size
0012                            *                  D1 :work register
0013
0014 005000 41F9 0000 5014                LEA       MEM_ST,A0       ;set start address to A0
0015 005006 7009                          MOVEQ     #10-1,D0        ;set loop counter to D0
0016 005008 7200                          MOVEQ     #0,D1           ;CLR D1
0017 00500A                     CLR_LOOP
0018 00500A 20C1                          MOVE.L    D1,(A0)+
0019 00500C 51C8 FFFC                     DBRA      D0,CLR_LOOP
0020
0022 005010 7000                          MOVEQ     #0,D0
0023 005012 4E40                          TRAP      #0
0024
0025                            *                  -----------
0026                            *                   data area
0027                            *                  -----------
0028
0029 005014 =000000A0           MEM_ST    DS.L      10*4
0030
0031                                      END
```


アプリケーション・ヒント

実際には，クリアすべき先頭と終了アドレスから操作すべきバイト数を算出し，ループカウンタと転送サイズを求める，というような処理が要求される．

図2.2 FILLの様子

2 連続したメモリ領域を順に0～255までの数値で満たす

●サンプルプログラム ［FILLC］

先の内容の変形であり，指定ブロックに０～255までの数値を順に書き込むものです．このようなパターンは，データ通信やCRTのテストパターンであるとか，ディスク・インターフェースのデバッグに有効です．

■動作

FILLCというアドレスから順に０，１，２.... 255，を書き込む．

■要点

書き込みアドレスを保持するアドレスレジスタと，書き込みパターンを保持するデータレジスタが必要であり，１回の書き込み終了後，それぞれ１だけ増加させれば目的を達成できる．

■レジスタの割り当て

A０：書き込みアドレスを保持し，１回の書き込みでインクリメント（＋１）する．

D０：何回書き込みをするかというループカウンタ値を保持し，１回の書き込みでデクリメント（－１）する．

D１：書き込みパターンを保持し，１回の書き込みで次のパターンに更新（＋１）する．

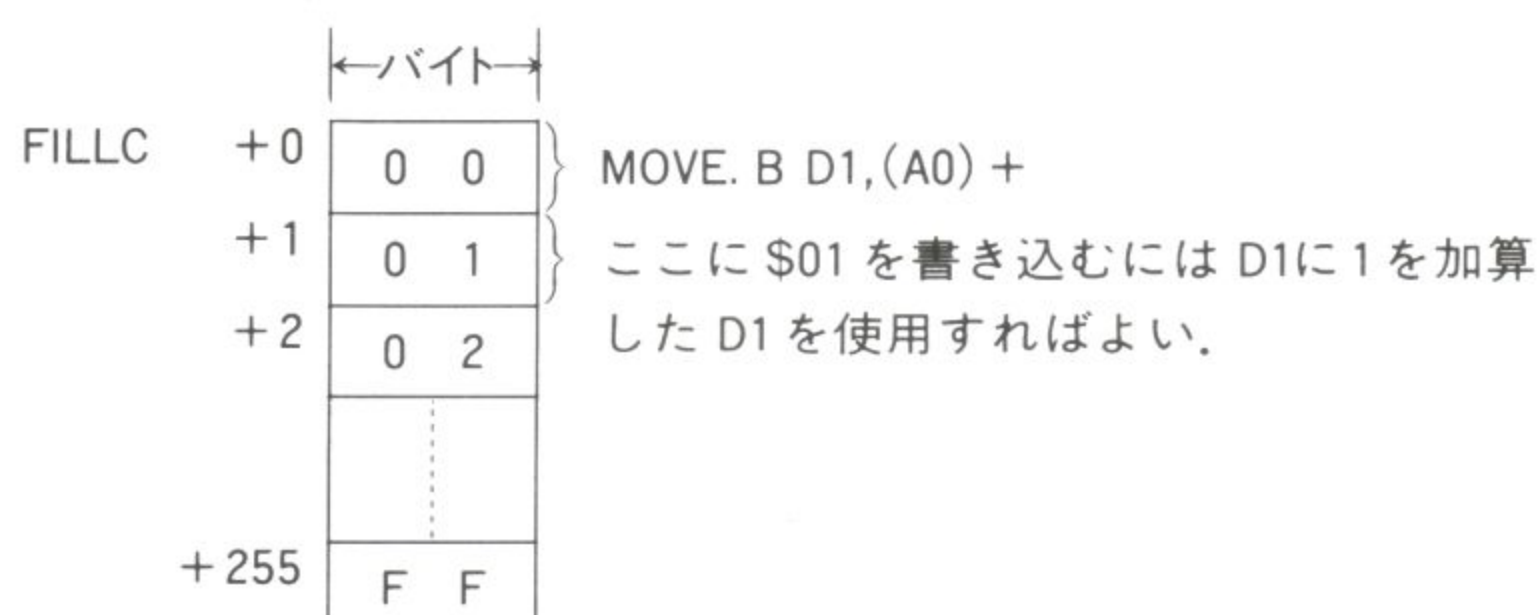
リスト［FILLC］

```
LINE ADDR. CODE/VALUE         LABEL     OP        OPERAND          COMMENT

0005
0006          =00005000                 ORG       $5000
0007          =00005000
0008 005000 41F9 0000 5018              LEA       FILLC,A0         ;pointer
0009 005006 303C 00FF                   MOVE.W    #256-1,D0        ;loop counter
0010 00500A 7200                        MOVEQ     #0,D1            ;work(patern)
0011 00500C                   FC_LOOP
0012 00500C 10C1                        MOVE.B    D1,(A0)+
0013 00500E 5201                        ADDQ.B    #1,D1
0014 005010 51C8 FFFA                   DBRA      D0,FC_LOOP
0015
0017 005014 7000                        MOVEQ     #0,D0
0018 005016 4E40                        TRAP      #0
0019
0020                          *                  ---------
0021                          *                  data area
0022                          *                  ---------
0023
0024 005018 =00000100         FILLC     DS.B      256
0025
0026                                    END
```

アプリケーション・ヒント

連続した数値ではなく順に偶数だけとか奇数だけを書き込むとか，連続したアドレスではなく１つおきとか２つおきのメモリへの書き込みなども，同様な考え方で達成できる．

図2.3　FILLCの様子

		←バイト→	
FILLC	＋0	0 0	MOVE. B D1,(A0)＋
	＋1	0 1	ここに $01 を書き込むには D1に１を加算した D1 を使用すればよい．
	＋2	0 2	
		⋮	
	＋255	F F	

3 メモリブロックの内容を別のメモリブロックへコピーする

◉サンプルプログラム［BMOVE］

ブロック転送はプログラムの様々な部分で要求され，スクリーンの内容を切り替えたいが，あとで切り替える前の状態に復帰させたいとか，小規模なケースでは，文字列の複写や文字列と文字列との連結など，に応用されます．

■動作

アドレスSOUを先頭する領域の内容を，アドレスDESTを先頭とする領域へコピーする．

■要点

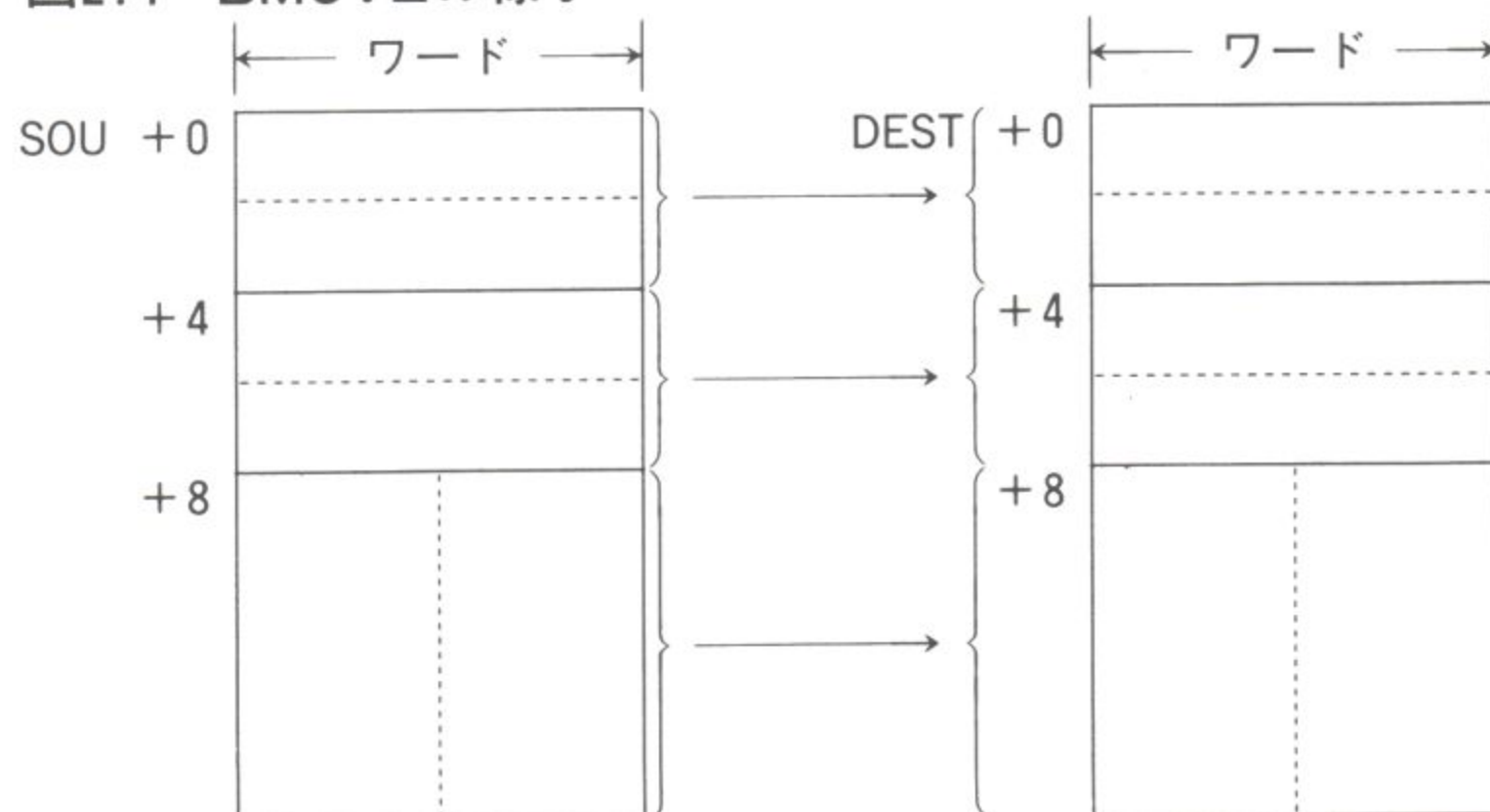
2つのメモリブロックがあるので，使用するアドレスレジスタが2つ必要であることと，転送命令の回数を指定するループカウンタとして，データレジスタが1つ必要である．

リスト［BMOVE］

```
LINE ADDR.  CODE/VALUE          LABEL    OP       OPERAND        COMMENT

0005
0006        =00005000                    ORG      $5000
0007        =00005000
0008                            *                 A0 :source address pointer
0009                            *                 A1 :destination address pointer
0010                            *                 D0 :length
0011
0012 005000 41F9 0000 5018               LEA      SOU,A0         ;set source address pointer to A0
0013 005006 43F9 0000 50B8               LEA      DEST,A1        ;set destination address pointer to A1
0014 00500C 7009                         MOVEQ    #10-1,D0       ;setup loop counter
0015 00500E                     BMOVE
0016 00500E 22D8                         MOVE.L   (A0)+,(A1)+
0017 005010 51C8 FFFC                    DBRA     D0,BMOVE
0018
0020 005014 7000                         MOVEQ    #0,D0
0021 005016 4E40                         TRAP     #0
0022
0023                            *                 ---------
0024                            *                 test area
0025                            *                 ---------
0026
0027 005018 =000000A0           SOU      DS.L     10*4
0028 0050B8 =000000A0           DEST     DS.L     10*4
0029
0030                                     END
```

アプリケーション・ヒント

無条件にブロックの内容をコピーするのではなく，データの内容を整理することもできる．つまり，読み込んだ内容をチェックし，書き込みをするか否かを決定し，書き込んだ要素の数もカウントする，という処理を想定できる．

図2.4　BMOVEの様子

4 メモリブロック間で各要素を交換する

◉サンプルプログラム　[EXG__AR]

配列（メモリブロックの各要素）の内容と別の配列の内容との交換は，データのソート（並び替え）や特定のブロックを別の内容で置き換える場合に要求されます．

■動作

配列ARRAY__Aと配列ARRAY__Bとの内容を，そっくり入れ換える．

■要点

単にコピーする処理では不要であったデータの保持（バックアップ）が必要で，交換という処理は以下の3つの過程が必要となる．

① 一方の要素（ARRAY__B）の内容をどこかに保持する．
② 空きになった場所（ARRAY__B）へ一方の要素（ARRAY__A）を転送．
③ 空きになった場所（ARRAY__A）へ一方の要素（ARRAY__B）を転送．

というように，データを書き込みから保護するために，書き込まれる場所の内容を一時的に退避した後で書き込む，ということになる．

アドレッシングモードであるが，一度読み込んだ後にアドレスレジスタが更新されては次の要素へ書き込んでしまうので，(An)＋を使うことはできず，手作業でアドレスを増加させることになる．

■各行の意味

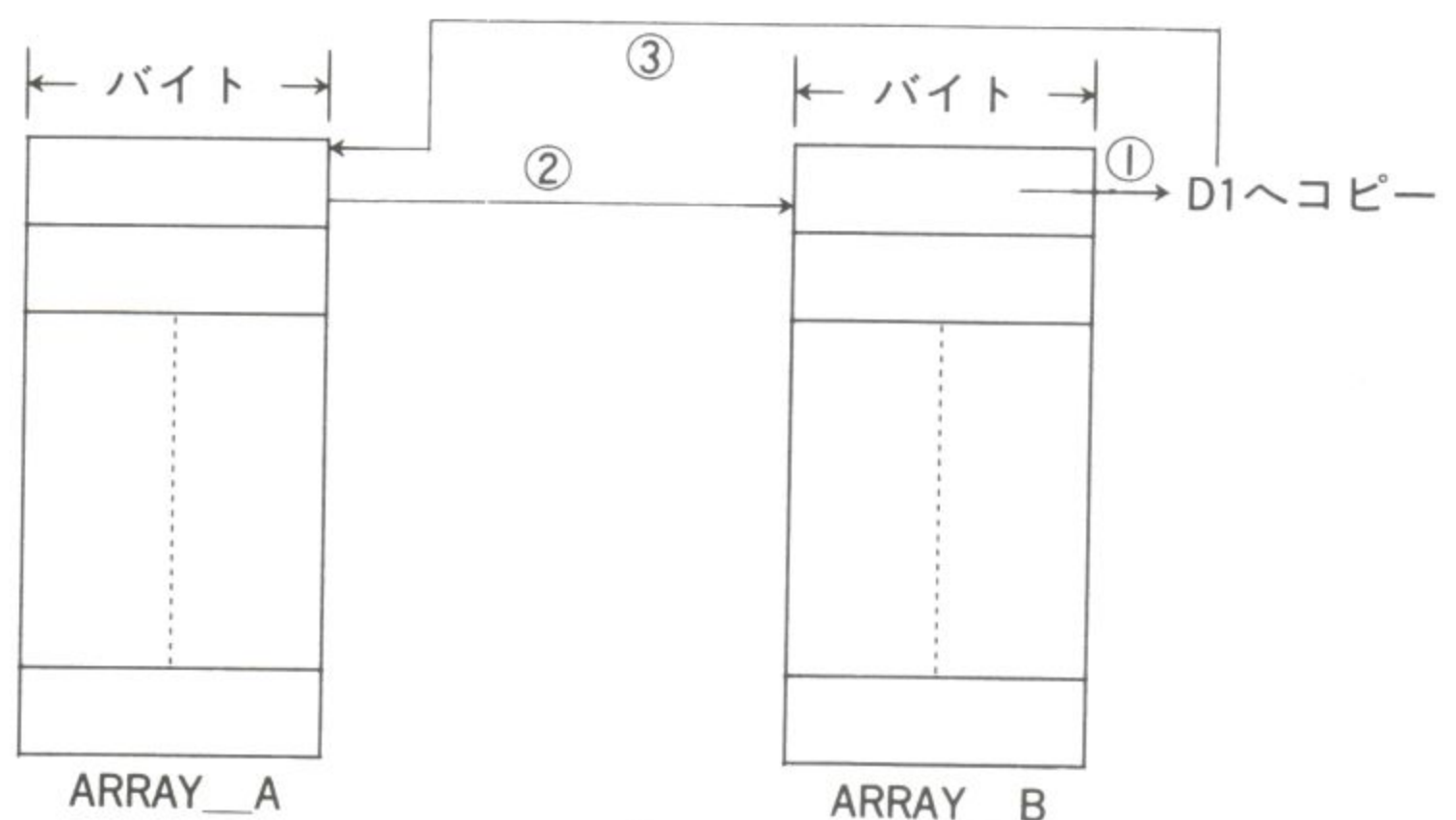
行13～15：必要なレジスタを初期化している．

行17～19：上記の3ステップにより，内容の交換をしている．

行21～23：双方のアドレスを更新する部分とループ制御を行っている．

アプリケーション・ヒント

ARRAY__Aの上半分をARRAY__Bの下半分と交換するとか，アドレスの操作を一方はプラス方向，他方はマイナス方向にすると，ARRAY__Aの先頭とARRAY__Bの最後とを対比して交換できる．

図2.5　EXG__ARの様子

リスト［EXG_AR］

```
LINE ADDR. CODE/VALUE            LABEL     OP       OPERAND         COMMENT

0005
0006        =00005000                      ORG      $5000
0007
0008                             *                A0 :address pointer of array_a
0009                             *                A1 :address pointer of array_b
0010                             *                D0 :length
0011                             *                D1 :work register
0012
0013 005000 41F9 0000 5022                 LEA      ARRAY_A,A0
0014 005006 43F9 0000 5122                 LEA      ARRAY_B,A1
0015 00500C 303C 00FF                      MOVE.W   #256-1,D0
0016 005010                      EXG_ARY
0017 005010 1211                           MOVE.B   (A1),D1
0018 005012 1290                           MOVE.B   (A0),(A1)
0019 005014 1081                           MOVE.B   D1,(A0)
0020
0021 005016 5288                           ADDQ.L   #1,A0           ;next
0022 005018 5289                           ADDQ.L   #1,A1
0023 00501A 51C8 FFF4                      DBRA     D0,EXG_ARY
0024
0026 00501E 7000                           MOVEQ    #0,D0
0027 005020 4E40                           TRAP     #0
0028
0029                             *                ---------
0030                             *                data area
0031                             *                ---------
0032
0033 005022 =00000100            ARRAY_A   DS.B     256
0034 005122 =00000100            ARRAY_B   DS.B     256
0035
0036                                       END
```

第2部 プログラム

5 構造体への要素をアクセスする

◉サンプルプログラム ［IX］

データの構造化にはディスプレースメント付きアドレスレジスタ間接形式によるアドレッシングが便利であり，アセンブラや逆アセンブラあるいはコンパイラなどの言語開発，ビジネス・アプリケーションなど，より複雑な処理をする分野へ応用されます．

■動作

ここではキャサリンという女性に関するデータを，名前，身長，バスト，ウエスト，ヒップ，に分類し，バストをBUST__DATという場所へ格納する．

■要点

① 各要素はベースアドレスからのオフセットでアクセスするので，あらかじめ構造体の内容を表現するだけの各要素のサイズをプログラマ自身が設定しておく．

② 各要素へのオフセットは，EQUという疑似命令でシンボル化するのが賢明であり，この定義は，アセンブラのニーモニックが記述される以前に一括して定義してしまう（ORGの前の方に置くこともできる）．

③ 一度ベース・アドレスをアドレスレジスタへ設定してしまえば，オフセットを指定するだけですべての要素を操作できる．

■各行の意味

行11〜15：構造体の５つの要素へのオフセットをシンボル化している．

行17〜19：３番目の要素であるバストをBUST__DATという領域へ格納する．

行18　：MOVE. B　21 (A0), BUST__DAT
と記述してもよいが，21をシンボルで指定した方がわかりやすい．

行19　：BUST__DATという領域を直接指定するので，絶対アドレッシングということになる．

行28　：012... という部分は，空白の個数を明示するためのスケールのつもりである．（名前欄に20バイト予約）．

アプリケーション・ヒント

実際には，大きな構造体の中に小さな構造体が存在することもある．

この例の要素に電話番号のフィールドを付加する場合，03－000－0000のように，さらに3つのフィールドで管理すれば，ある条件に従って取り出すことができる．

すでに気づかれたでしょうが，特定要素の平均を求めたり，各要素の範囲を設定し，この条件を満足する女性を選び出すとか，我々が日常お世話になっているデータベースの内部構造は，このようなものを意味する．

図2.6　IXの様子

リスト［IX］

```
LINE ADDR.  CODE/VALUE          LABEL     OP      OPERAND        COMMENT

0005
0006        =00005000                     ORG     $5000
0007
0008                            *               ------
0009                            *               struct
0010                            *               ------
0011        =00000000           NAME      EQU     0
0012        =00000014           HEIGHT    EQU     20
0013        =00000015           BUST      EQU     21
0014        =00000016           WAIST     EQU     22
0015        =00000017           HIP       EQU     23
0016
0017 005000 41F9 0000 5014                LEA     PERSONAL,A0
0018 005006 1028 0015                     MOVE.B  BUST(A0),D0
0019 00500A 13C0 0000 502C                MOVE.B  D0,BUST_DAT
0020
0022 005010 7000                          MOVEQ   #0,D0
0023 005012 4E40                          TRAP    #0
0024
0025                            *               ---------
0026                            *               data area
0027                            *               ---------
0028                            *                 01234567890123456789
0029 005014 6B61 7468 6172      PERSONAL  DC.B    "katharine           " ;name for 20 byte
            696E 6520 2020
            2020 2020 2020
            2020
0030 005028 A9                            DC.B    169            ;height for 1 byte
0031 005029 58                            DC.B    88             ;bust for 1 byte
0032 00502A 3C                            DC.B    60             ;waist for 1 byte
0033 00502B 5A                            DC.B    90             ;hip for 1 byte
0034
0035 00502C =00000001           BUST_DAT  DS.B    1
0036
0037                                      END
```

6
整数算術演算命令サンプルプログラム

1 1行のチェックサムを求める

◉サンプルプログラム ［LINESUM］

チェックサムとは，あるまとまった長さのデータを数値で表現するために使用され，完全ではないにしても，データ・ブロックのチェックに便利で，たとえば，ディスケットの内容をクラッシュしても，それが1部分であれば，チェックサムを比較することによって容易に破壊箇所を発見できます．

チェックサムの算出には，単に加算するもの，論理演算によるもの，これらの2つの組み合わせたものなどがあり，ディスケットが512バイト／セクタなら，1行を16バイトと考えて1つのチェックサムを求め，さらに各行（32行）のチェックサムをトータル・チェックサムとしたり，横16／縦32 のマトリックスと考え，横方向と縦方向のチェックサムとトータル・チェックサムを算出することもできます．

■動作

SUM__STというアドレスから16バイトのチェックサムを求めるが，ワード（2バイト）の単純加算とし，結果をRESULTというアドレスへ格納する．

■要点

① 加算へ入る前にＤ０をクリアしておき，Ｄ０へ結果を格納する．

② (An)＋によって次々にデータの格納されるアドレスをポイントする．

③ 予定は８ワード（16バイト）なので，８回のワード加算を行うため，ループ回数には７（８－１）をＤ１に指定する．

■各行の意味

行６～７：定数の定義をしているが，７行の式ではこのようにも記述できるというだけのことで，R__COUNTというシンボルを７という数値として使いたいわけである．

行12～15：必要なレジスタを初期化している．

行13　　：次の表現に注意する必要がある．

```
MOVE.W #R__COUNT,D1   単なるデータであるR__COUNTがＤ１へ転送される
MOVE.W  R__COUNT,D1   アドレスR__COUNTの内容がＤ１へ転送される
```

R__COUNT自体は７という値に設定しているので，#R__COUNTは７という即値を意味し，R__COUNTなら７番地のメモリの内容をＤ１に転送するが，奇数番地に対するワードアクセスは残念ながらアドレスエラーとなる．

行20　　：結果を１箇所しか格納しないなら，次のように直接格納先を指定することもできる．

```
MOVE.W D0, RESULT
```

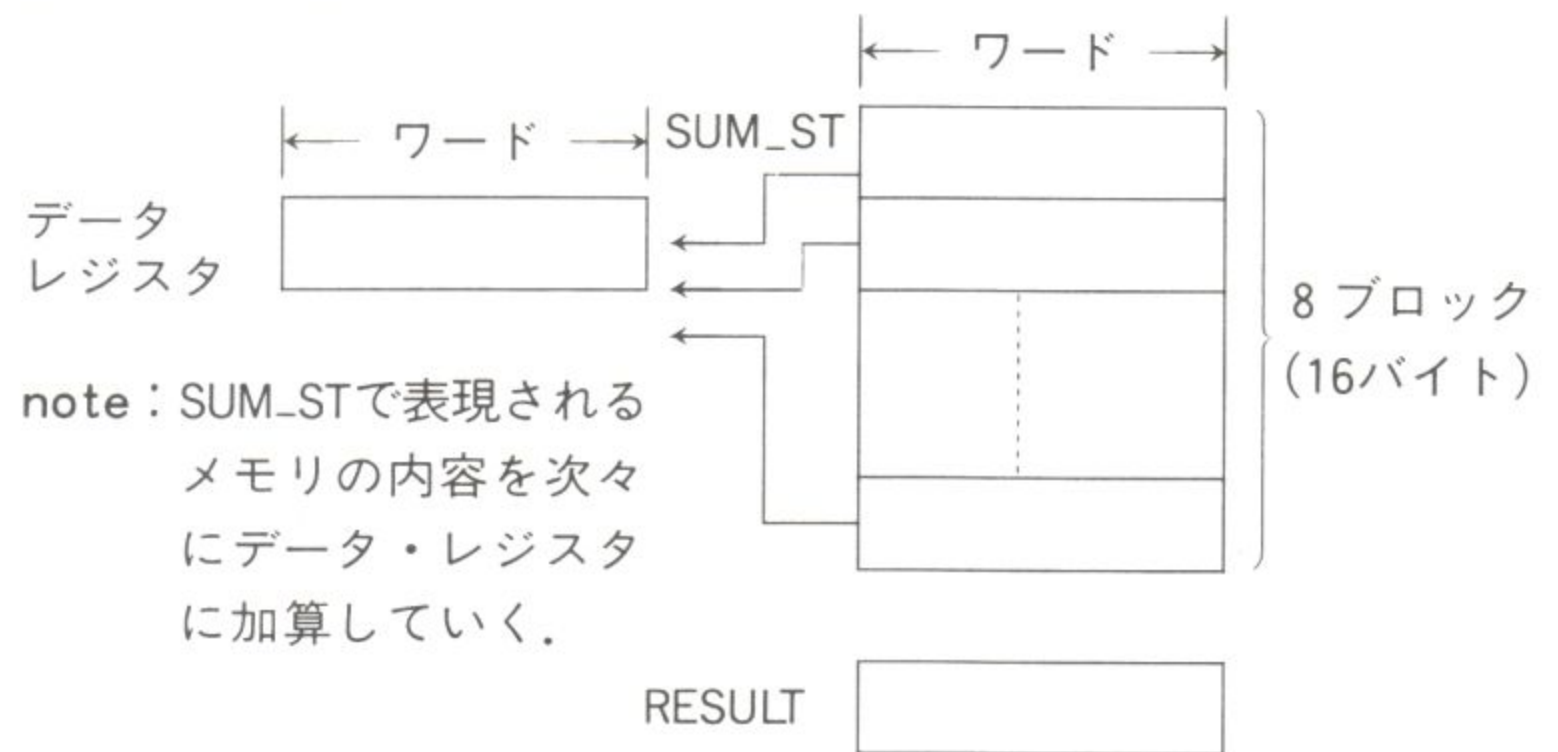
アプリケーション・ヒント

加算対象となるデータが3番地おきに格納されるようなデータ構造など，データが隣接しているとは限らない．この場合には手作業でアドレスを進めてやる必要がある．

つまり，

ADDQ. L #DISTANCE, An

のようにして，次の対象となるデータが格納されているアドレスへ，アドレスレジスタを進める．

図2.7　LINESUMの様子

リスト［LINESUM］

```
LINE ADDR.  CODE/VALUE          LABEL     OP      OPERAND          COMMENT

0005
0006          =00000002         WORD_LEN  EQU     2
0007          =00000007         R_COUNT   EQU     (16/WORD_LEN)-1
0008
0010          =00005000                   ORG     $5000
0011
0012 005000 4240                          CLR.W   D0
0013 005002 323C 0007                     MOVE.W  #R_COUNT,D1      ;setup loop counter
0014 005006 41F9 0000 5020                LEA     SUM_ST,A0
0015 00500C 43F9 0000 501E                LEA     RESULT,A1
0016 005012                     SUM_LOOP
0017 005012 D058                          ADD.W   (A0)+,D0
0018 005014 51C9 FFFC                     DBRA    D1,SUM_LOOP
0019
0020 005018 32C0                          MOVE.W  D0,(A1)+
0022 00501A 7000                          MOVEQ   #0,D0
0023 00501C 4E40                          TRAP    #0
0024
0025 00501E =00000002           RESULT    DS.W    1
0026 005020 =00000010           SUM_ST    DS.W    16/WORD_LEN
0027
0028                                      END
```

2 64ビット(8バイト)データの加算を行う

◉サンプルプログラム ［ADD64］

通常の演算範囲としては32ビット（ロングワード）もあれば十分であり，これ以上の演算が要求されることは稀です．しかし，外部から入力するパルスを長時間監視するなど，実験データの収集には必要かもしれません．

■動作

いずれも64ビット長のメモリレジスタ０（MR0）とメモリレジスタ１（MR1）とを加算し，結果をメモリレジスタ（MR1）へ格納するが，MR0の内容は保持される．

■要点

① データレジスタを２つ連結してもよいが，メモリを連結すればデータサイズは無制限に拡張できることから，ここではMR0, MR1を先頭とする８バイトのメモリ空間をレジスタと想定して加算を行う．

② メモリ空間に存在するオペランドの多倍精度加算なので，選択すべきアドレッシングは，

```
ADDX  −(An), −(An)
```

となる（データがどのように記憶されるか，確認してほしい）．

③ 標準では32ビットの加算が許されているだけで，これ以上の演算には手作業が伴い，Xビットと Zビットの扱いに注意が必要である．

Xビット：演算直前でクリア（０）する

Zビット：演算直前でセット（１）する

このフラグは演算結果がゼロであることを示すものだが，次のように行動する．

結果＝０　<u>変化しない</u>

結果≠０　クリア

通常は結果がゼロであることを示すためにセットされることを期待するが，以上のように<u>変化しない</u>ので，プログラマ自身があらかじめ Zビットをセットしなければならない．

64ビットの加算では，32ビットの下位どうし次にXビットを含めた32ビットの上位どうし，の順に加算するので，双方の64ビット値がゼロであれば，演算前に手作業で操作したZビットが保存され，そうではなく，下位32ビットであれ上位32ビットであれ，双方のいずれかの32ビット値がゼロでなければ，加算を実行した時点で（結果≠０）Zビットはクリアされるはずである．

このようにしてZビットは多倍精度の演算結果を正しく反映する．

■加算過程

① Zビットだけをセットし他の全ビットをクリアする（ユーザ状態で実際に操作できるのは，X，N，Z，V，C，だけ）．

② （下位32ビット）＋（下位32ビット）を行う

③ （上位32ビット）＋（上位32ビット）＋（Xビット）を行う

■各行の意味

行６　：コメントであり，７行のビット番号がどのようになっているかを明示するために記述した（Ｚビットだけがセットされていることがよくわかる）．

行７〜９：必要な定数を定義しているが，７行のオペランドの先頭の〝%〟は，２進数であることをアセンブラへ知らせる．

行14〜16：必要なレジスタを初期化している．

行15，16：−（An）を使用してメモリレジスタを操作するので，最下位桁の更に先へアドレスレジスタを初期化している．初期化したアドレス値ではなく，４だけ減算したアドレスの内容（サイズはロングワード）が操作対象になる．

行18　：CCRへ即値を転送することで，ＸビットやＺビットなどを手作業で操作しているが，14行のようなCCRを変化させるような命令を本行の後に置くと，フラグが変化することになり，正しい結果が得られないことになる．
つまり，18行はここが指定席であり，必ず演算直前に記述しなければならない．

行19〜21：32ビット値を２回に分けて加算している．

アプリケーション・ヒント

メモリレジスタ長を必要なビットサイズにも拡張できるし，データレジスタを連結し，多倍精度演算を行うことも容易である．

図2.8 ADD64の様子

64ビット表現

	63 … 32	31 … 0
MR0	4 E 7 2 4 1 1 8	0 0 5 A 7 2 0 1
MR1	C 1 C A 0 D 4 A	2 8 4 0 5 0 1 9

ロングワードでメモリへ記憶される様子

		ロングワード
MR0	+0	4 E 7 2 4 1 1 8
	+4	0 0 5 A 7 2 0 1
初期化されたA0→	+8	
MR1	+0	C 1 C A 0 D 4 A
	+4	2 8 4 0 5 0 1 9
初期化されたA1→	+8	

加算の様子

	上位32ビット	下位32ビット	
	4 E 7 2 4 1 1 8	0 0 5 A 7 2 0 1	MR0
	C 1 C A 0 D 4 A	2 8 4 0 5 0 1 9	MR1
+	1	0	0 Xビット
	1 0 3 C 4 E 6 2	2 8 9 A C 2 1 A	MR1（結果）

note：アドレスレジスタはベースアドレスから＋８した地点に初期化する．このようにしないと，加算データをアクセスできない．

リスト[ADD64]

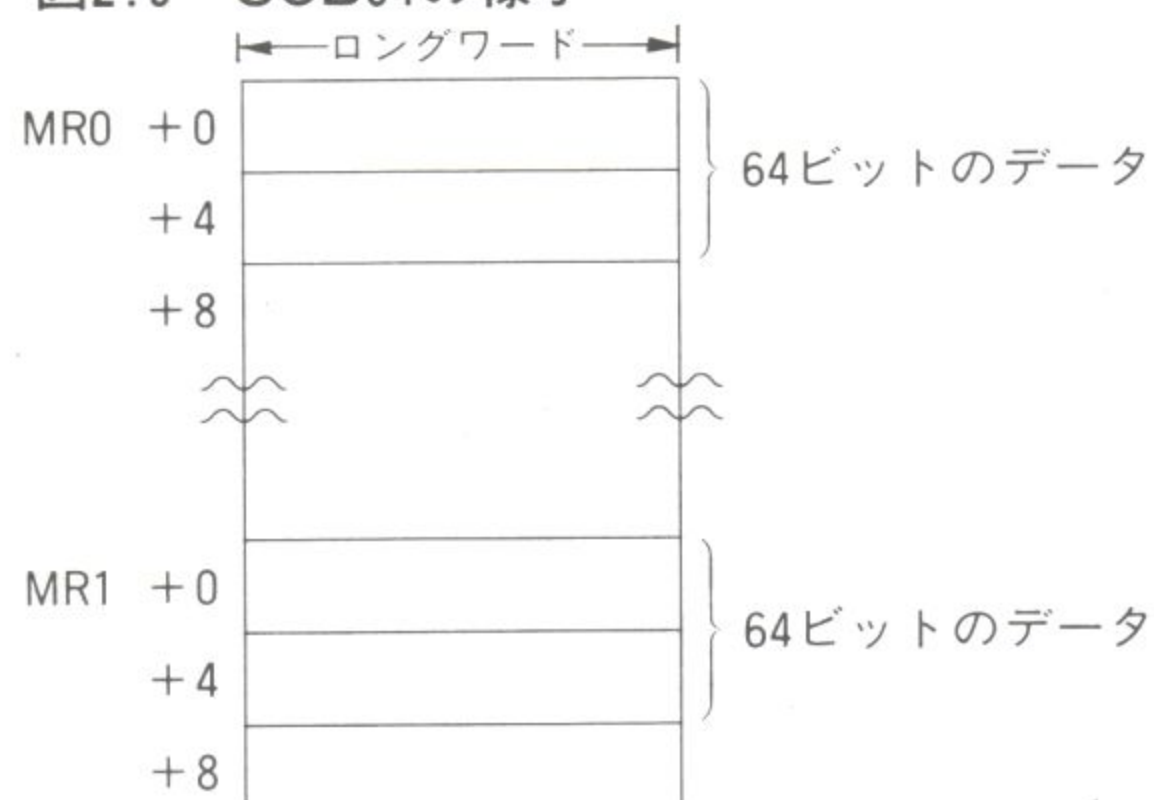
```
LINE ADDR.  CODE/VALUE          LABEL     OP        OPERAND          COMMENT

0005
0006                            *                    76543210765XNZVC
0007          =00000004         X_CLR     EQU       %0000000000000100
0008          =00000008         B64_CNT   EQU       8
0009          =00000001         ADD_CNT   EQU       2-1
0010
0012          =00005000                   ORG       $5000
0013
0014 005000 7001                          MOVEQ.L #ADD_CNT,D0      ;set loop count to D0
0015 005002 41F9 0000 5024                LEA       MR0+B64_CNT,A0 ;set memory register_0 pointer to A0
0016 005008 43F9 0000 502C                LEA       MR1+B64_CNT,A1 ;set memory register_1 pointer to A1
0017
0018 00500E 44FC 0004                     MOVE.W    #X_CLR,CCR       ;clear X-bit,set Z-bit
0019 005012                     ADD_LOOP
0020 005012 D388                          ADDX.L    -(A0),-(A1)      ;64 bit addtion
0021 005014 51C8 FFFC                     DBRA      D0,ADD_LOOP
0022
0024 005018 7000                          MOVEQ     #0,D0
0025 00501A 4E40                          TRAP      #0
0026
0027                            *                  --------------------
0028                            *                  memory register area
0029                            *                  --------------------
0030
0031 00501C =00000008           MR0       DS.L      2                ;64 bit(32*2)
0032 005024 =00000008           MR1       DS.L      2                ;64 bit(32*2)
0033
0034                                      END
```

3 64ビット(8バイト)データの減算を行う

◉サンプルプログラム ［SUB64］

説明するまでもなく，ADD64でのADDX.L がSUBX.Lになっただけです．

図2.9　SUB64の様子

|←ロングワード→|
MR0 +0
+4
+8
MR1 +0
+4
+8
64ビットのデータ
64ビットのデータ

リスト［SUB64］

```
LINE ADDR. CODE/VALUE          LABEL     OP        OPERAND          COMMENT

0005
0006                           *                    76543210765XNZVC
0007         =00000004         X_CLR     EQU       %0000000000000100
0008         =0Q000008         B64_CNT   EQU       8
0009         =00000001         SUB_CNT   EQU       2-1
0010
0012         =00005000                   ORG       $5000
0013
0014 005000 7001                         MOVEQ.L  #SUB_CNT,D0      ;set loop count to D0
0015 005002 41F9 0000 5024               LEA      MR0+B64_CNT,A0   ;set memory register_0 pointer to A0
0016 005008 43F9 0000 502C               LEA      MR1+B64_CNT,A1   ;set memory register_1 pointer to A1
0017
0018 00500E 44FC 0004                    MOVE.W   #X_CLR,CCR       ;clear X-bit,set Z-bit
0019 005012                    SUB_LOOP
0020 005012 9388                         SUBX.L   -(A0),-(A1)      ;64 bit substraction
0021 005014 51C8 FFFC                    DBRA     D0,SUB_LOOP
0022
0024 005018 7000                         MOVEQ    #0,D0
0025 00501A 4E40                         TRAP     #0
0026
0027                           *                 --------------------
0028                           *                 memory register area
0029                           *                 --------------------
0030
0031 00501C =00000008          MR0       DS.L     2                ;64 bit(32*2)
0032 005024 =00000008          MR1       DS.L     2                ;64 bit(32*2)
0033
0034                                     END
```

4 符号なし32ビット(4バイト)データの乗算を行う

◉サンプルプログラム ［MUL32］

68000は内部レジスタが32ビット構成であるにもかかわらず，32ビット×32ビットの乗算はサポートされませんが，アセンブラのターゲットとなるプログラムでは，倍精度の乗除算が要求されることは稀です．

■動作

Ｄ０×Ｄ１　を計算し，上位32ビットはＤ０へ，下位32ビットはＤ１へ転送する．

■要点

① 符号なし乗算命令　MULU を使うが，サポートされるビット長は16ビットなので，上位16ビットと下位16ビットに分けて乗算を行い，これらを加算して64ビット長の結果を得る．

② Ｄ０の上位／下位をａ／ｂ，Ｄ１の上位／下位をｃ／ｄと考えると，４通りの積が得られる．

上位〜上位　ａ×ｃ
交差積　ａ×ｄ，ｃ×ｂ
下位〜上位　ｂ×ｄ

これらはMULUで求めることができるが，いずれも32ビット長であることと，図に示すように，互いに16ビット（ａ，ｂ，ｃ，ｄのサイズ）ずつオーバーラップしているので，単純にＸビットを考慮した32ビットの加算では，正しい結果は得られない．

③ 途中でＸビットを含めた加算（ADDX）を実行することになるが，この命令セットは貧弱であり（オペランドが限定されている），メモリをレジスタ（アキュムレータ）として使用するには無理があるので，本例のように，豊富なデータレジスタを操作した方がよい．

④ Ｄ２〜Ｄ４までのデータレジスタを結果の格納以外に使用する．

■各行の意味

行13〜16：16ビットのデータに分離する部分．

行18〜22：（ａ×ｂ)×（ｃ×ｄ）を展開している部分で，<cross>とコメントされている項は，下位〜上位か上位〜下位による組み合わせ項である．
MULUのソース側が次のMULUのディスティネーション（結果の格納先）になっているが，このようにすると無駄なレジスタを使用しなくてよい．

行23：Ｄ０とＤ３の内容を交換するが，これは，Ｄ０とＤ１を結果の格納先に使用するので，Ｄ０には上位項どうし，Ｄ１には下位項どうしの部分積を格納した方が便利だからである．

行24〜29：これまでの結果をコメントとして列記している．
この結果は，積の下位32ビットの上位ワード（16ビット）と，上位32ビットの下位ワード（16ビット）に重なっている．

行30　：<cross>項である交差積を加算する.

行32～34：30行の結果を下位32ビットに分離（30行のXビットを保持).
CLR.Wは，指定オペランドの下位16ビットだけをクリアする.

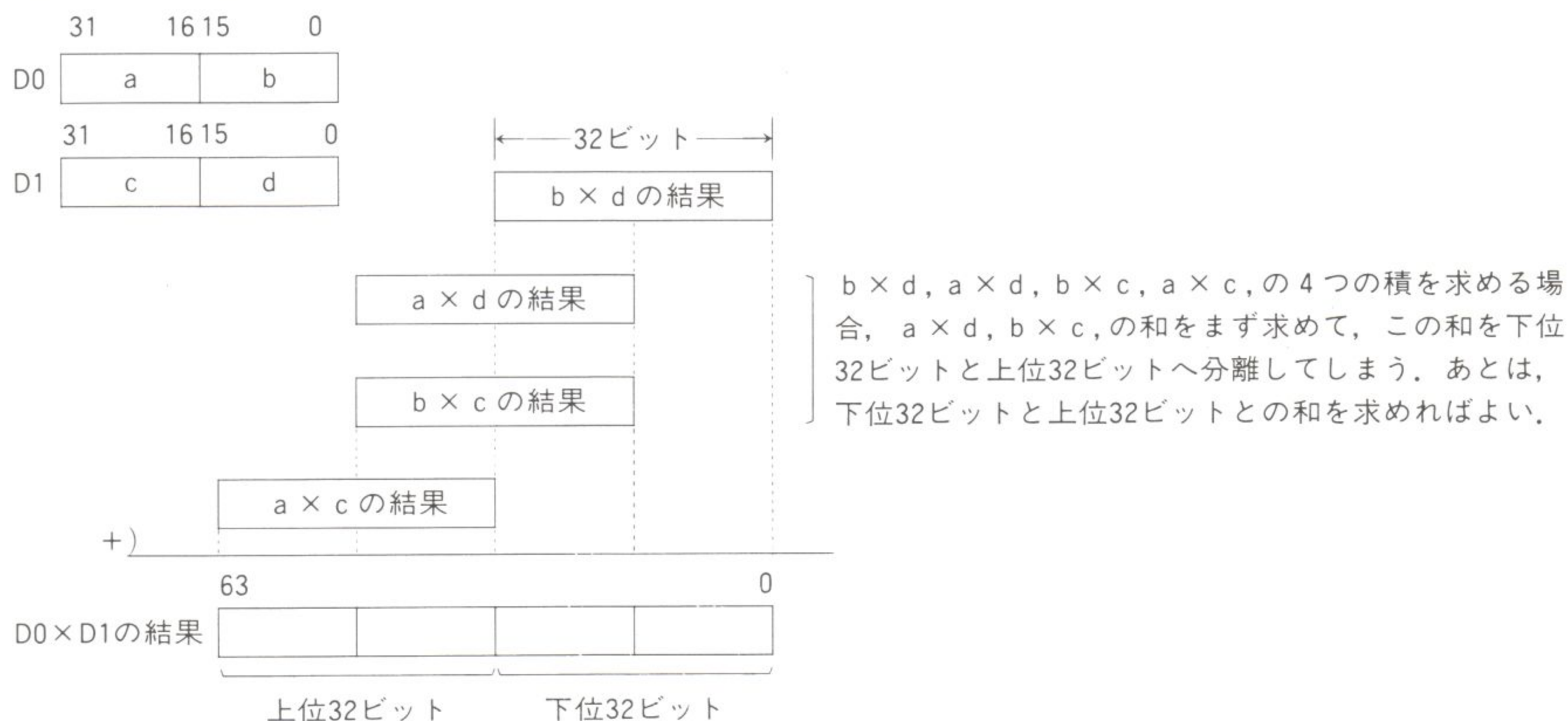
行36～38：30行のXビットを考慮しつつ，30行の結果を上位32ビットに分離.

行40～41：（下位32ビット＋下位32ビット)＋（上位32ビット＋上位32ビット)＋Xビットを求めるが，ADDXをいきなり使用しないのであれば，XビットやZビットを操作しなくてよい．つまり，CCRへの転送は不要となる.

アプリケーション・ヒント

多倍精度の符号付き2進乗算は，MULS命令による部分積を加算するアルゴリズムでは不可能であり，乗数，被乗数の絶対値をとり(負なら正になおす)，MULUで積を求め，さらに符号処理をしなければならない.

ただし，前述にように，アセンブラでのプログラミングでは，多倍精度の乗除算が要求されることは非常に少なく，MULUやMULSで処理できる分野がほとんどである，と考えてもよく，符号を考慮しなければならないことも少ないと思われる.

図2.10　MUL32の様子

リスト[MUL32]

```
LINE ADDR. CODE/VALUE          LABEL    OP       OPERAND         COMMENT

0005
0007         =00005000                  ORG      $5000
0008
0009                           *               32 bit unsigned multiplication
0010                           *
0011                           *               D0.L * D1.L --> D0-D1
0012
0013 005000 2400                        MOVE.L   D0,D2           ;D0.W = b
0014 005002 2601                        MOVE.L   D1,D3           ;D1.W = d
0015 005004 4842                        SWAP     D2              ;D2.W = a
0016 005006 4843                        SWAP     D3              ;D3.W = c
0017 005008
0018 005008 3801                        MOVE.W   D1,D4           ;save d to D4
0019 00500A C2C0                        MULU     D0,D1           ;D1.L = b*d
0020 00500C C0C3                        MULU     D3,D0           ;D0.L = c*b  <cross>
0021 00500E C6C2                        MULU     D2,D3           ;D3.L = a*c
0022 005010 C4C4                        MULU     D4,D2           ;D2.L = d*a  <cross>
0023 005012 C143                        EXG      D0,D3
0024                           *
0025                           *               D0.L = a*c  upper 32 bit
0026                           *               D1.L = b*d  lower 32 it
0027                           *               D2.L = d*a  <cross>
0028                           *               D3.L = c*b  <cross>
0029                           *
0030 005014 D483                        ADD.L    D3,D2           ;D2.L = (ad+bc)
0031
0032 005016 3602                        MOVE.W   D2,D3
0033 005018 4843                        SWAP     D3
0034 00501A 4243                        CLR.W    D3              ;D3.L = (ad+bc)L,0000
0035
0036 00501C 4242                        CLR.W    D2
0037 00501E D543                        ADDX.W   D3,D2           ;add X bit
0038 005020 4842                        SWAP     D2              ;D2.L = 000X,(ad+bc)H
0039
0040 005022 D283                        ADD.L    D3,D1           ;lower order 32 bit
0041 005024 D182                        ADDX.L   D2,D0           ;high  order 32 bit
0042
0044 005026 7000                        MOVEQ    #0,D0
0045 005028 4E40                        TRAP     #0
0046
0047                                    END
```

5 符号なし32ビット÷16ビットの除算を行う

◉サンプルプログラム ［DIV］

■動作

D__AREAから連続したメモリ領域に存在する16ワードを，DV__DATAで割り，その余りをREM__AREA から連続したメモリ領域へ格納する．

■要点

① 除算命令のディスティネーション部（ここではメモリ上のデータ）は32ビットに限定されるので，16ビットで十分であっても，上位ワードにゼロを満たした32ビット値として扱う．

② 結果はディスティネーションで指定したDnへ格納され，メモリ上に置くことは許されない．

③ 商や余りの格納には16ビットを必要とするので，これらが16ビットで表現できる範囲にならなければならないが，アセンブラの適用分野では，扱う値の範囲が不明であることは稀で，汎用の演算ルティーンに要求されるこのような心配を，さほどしなくてもよいケースがほとんどである．
ただし商や余りの範囲を的確に把握しておかねばならないのは，いうまでもないことである．

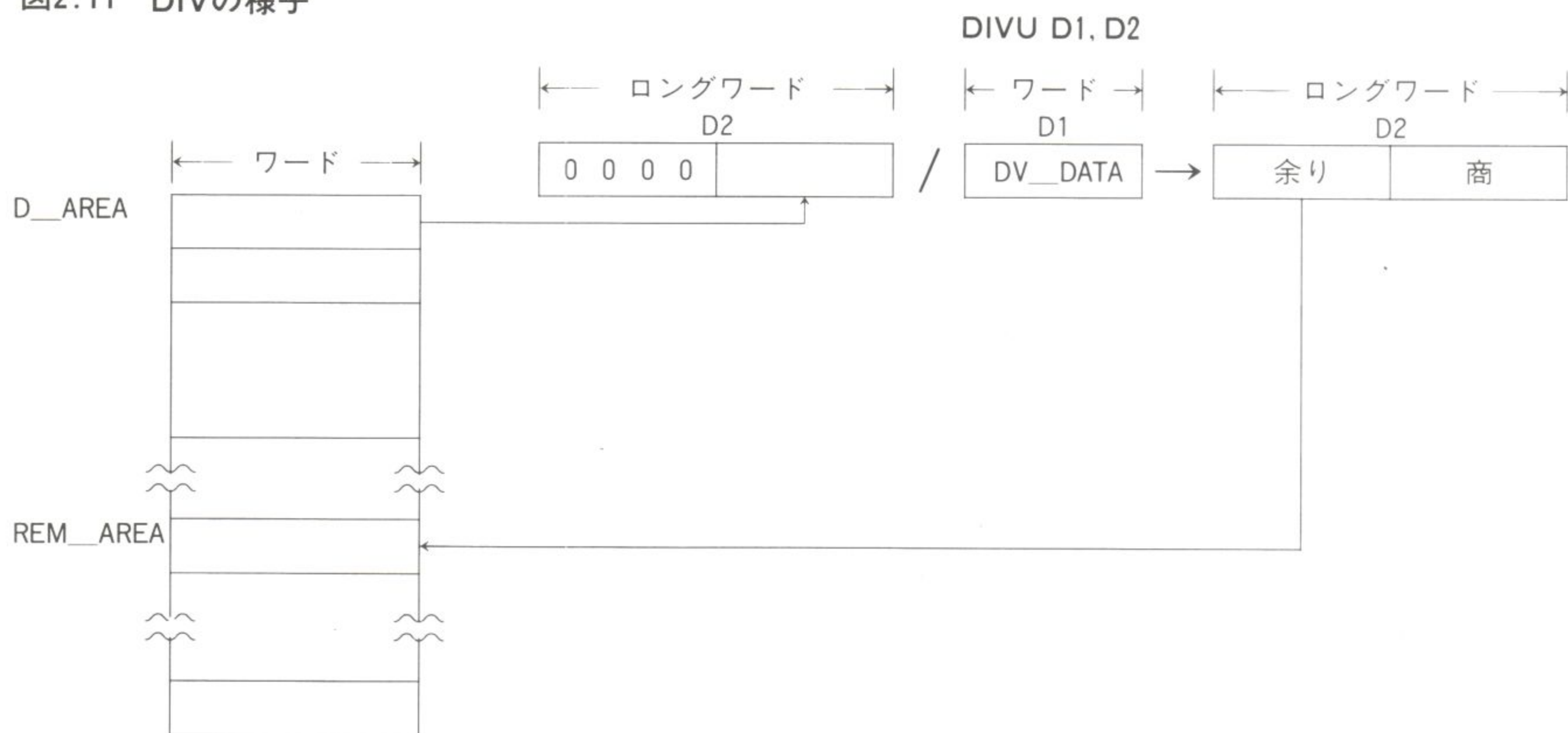
④ 余りはディスティネーションで指定したDnの上位ワードへ格納されるので，余りだけを操作するのであれば，SWAP命令を実行しなければならない．

■各行の意味

行６～７：必要な定数を指定．

行12～15：必要なレジスタを初期化している．
この時点で　Ｄ２／Ｄ１　を行うが，Ｄ２の下位ワードはメモリから，Ｄ１には定数をセットすることを想定している．

行17～22：必要な処理を行っている．

図2.11　DIVの様子

リスト[DIV]

```
LINE ADDR. CODE/VALUE           LABEL     OP       OPERAND          COMMENT

0005
0006          =00000101         DV_DATA   EQU      257
0007          =00000010         AR_LEN    EQU      16               ;16 word
0008
0010          =00005000                   ORG      $5000
0011
0012 005000 700F                          MOVEQ    #AR_LEN-1,D0     ;setup loop counter
0013 005002 323C 0101                     MOVE.W   #DV_DATA,D1
0014 005006 41F9 0000 5024                LEA      D_AREA,A0
0015 00500C 43F9 0000 5044                LEA      REM_AREA,A1
0016 005012                     DIV_LOOP
0017 005012 7400                          MOVEQ    #0,D2            ;clear D2
0018 005014 3418                          MOVE.W   (A0)+,D2
0019 005016 84C1                          DIVU     D1,D2            ;D2.L/D1.W --> D2.L
0020 005018 4842                          SWAP     D2
0021 00501A 32C2                          MOVE.W   D2,(A1)+
0022 00501C 51C8 FFF4                     DBRA     D0,DIV_LOOP
0023
0025 005020 7000                          MOVEQ    #0,D0
0026 005022 4E40                          TRAP     #0
0027
0028 005024 =00000020           D_AREA    DS.W     AR_LEN
0029 005044 =00000020           REM_AREA  DS.W     AR_LEN
0030
0031                                      END
```

6 メモリ上から特定の値をサーチする

◉サンプルプログラム ［SDATA］

比較命令のサンプルですが，コンピュータが，あたかも頭脳をもっているように行動できる秘密の１つは，比較ができるからであり，使用頻度の高い命令です．

■動作

メモリ上のST__ADR～END__ADRまでの区間に，Ｄ０へセットしたパターンが存在するかを調べ，結果をＺビットに求める．

■要点

① 「どのようにして比較作業を終了するか」ということが先決であり，終了条件を的確に把握する必要がある（終了条件が複数であるケースも多い）．
② 指定パターンが見つかれば検索を終了し，見つからない場合でも最終アドレスまで到達すると作業を終了する．
③ 最終アドレスの内容は，ループ外での比較となる．
④ 結果をCCRのＺビットに求める．

表2.3

CCR	意　　味	アセンブラ表現
Ｚ＝１	指定パターンは存在する	EQ
Ｚ＝０	指定パターンは存在しない	NE

■各行の意味

行11～13：Ｄ０にパターン(通常はキーと呼ぶ)，Ａ０に読み出し側アドレス，Ａ１に最終アドレスをセット

行14～16：先頭～最終アドレスの直前までの検索を行い，見つかった時点でFOUNDへ制御を移行する．

行20～22：16行の比較結果がnot equalのときにここへ制御が移る．
最終アドレスまで到達したのかどうかをチェックする．
注意しなければならないのは，15行のＡ０は次のアドレスをポイントしていることである．
　Ａ１＞Ａ０；まだ最終アドレスまで更新されていないので，CMP__LOOPへ分岐する．
　Ａ１＝Ａ０；Ａ１＞Ａ０　に対する裏条件はＡ１≦Ａ０であり，そのうちの〝＝〟が本行である．
　15行では，最終アドレスの直前まで比較が実行され，その結果，最終アドレスの内容を比較する必要があることを意味する（Ａ０は最終アドレスを保持している）．

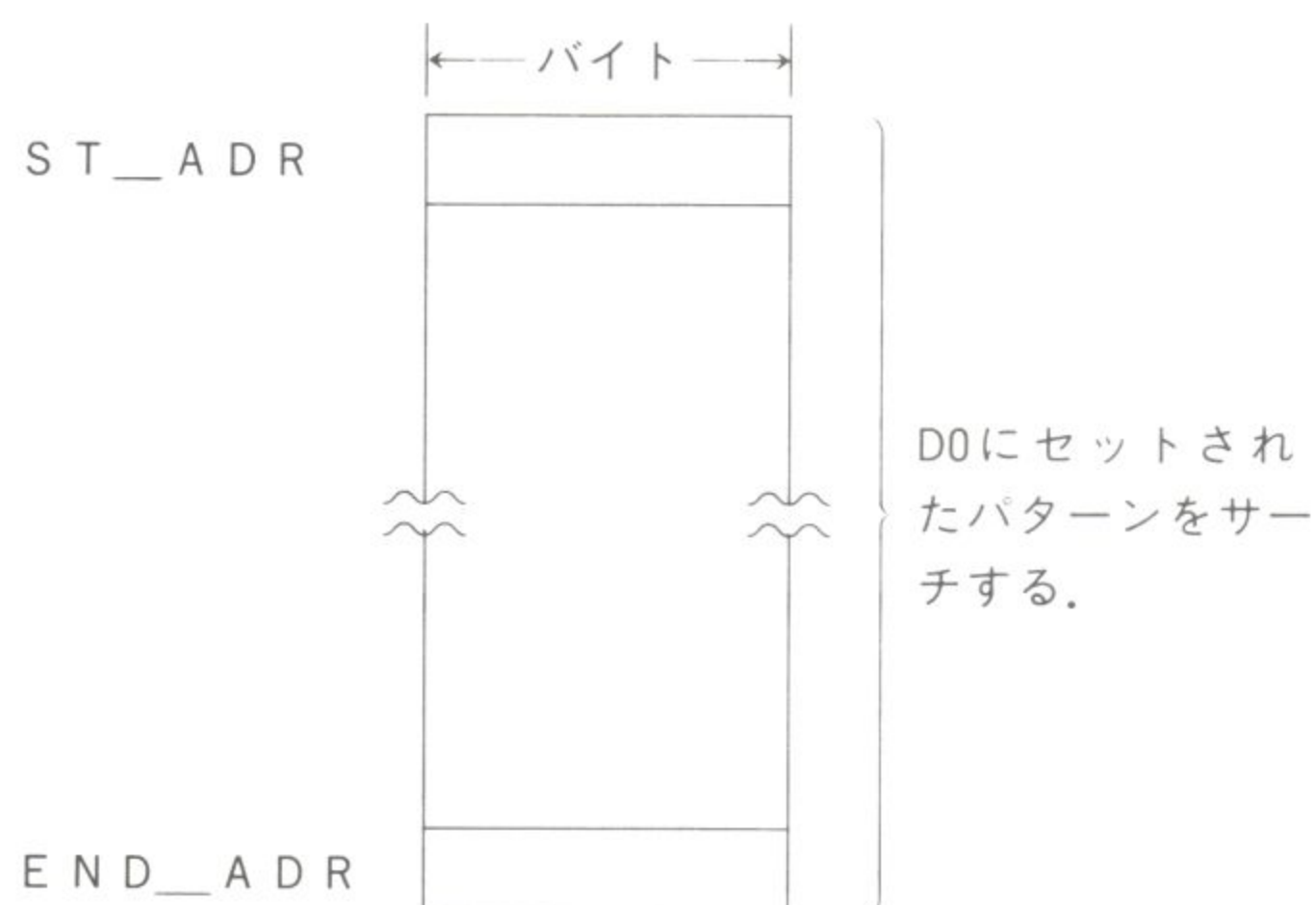
行23　　：CCRのＺビットに比較結果が求められる．

アプリケーション・ヒント

DBRAでは探索範囲が限定されてしまうが，最終アドレスをチェックすれば，どこまでも探索することができる．見つかったアドレスの条件をそろえるには，22行の次にADDAにより，比較サイズである1(イチ)をA0に加算するか，あるいは22行の(A0)を(A0)+にする．

このようにして，FOUNDでは，常に見つかったアドレスの次のアドレスとなるが，ADDAはCCRを変更しないので，最終アドレスの内容の比較結果は正しくZビットへ反映される．

本例では，一致したパターンが他にあるか否かを無視したが，実際には，一致したアドレスや一致した数を知りたいこともある．

図2.12　SDATAの様子

リスト[SDATA]

```
LINE ADDR.  CODE/VALUE          LABEL     OP       OPERAND          COMMENT

0005
0006        =0000005A           PATERN    EQU      $5A
0007
0009        =00005000                     ORG      $5000
0010
0011 005000 103C 005A                     MOVE.B   #PATERN,D0       ;set patern to D0
0012 005004 41F9 0000 501E                LEA      ST_ADR,A0        ;set address pointer to A0
0013 00500A 43F9 0001 501D                LEA      END_ADR,A1       ;set end address pointer to A1
0014 005010                     CMP_LOOP
0015 005010 B018                          CMP.B    (A0)+,D0
0016 005012 6706                          BEQ      FOUND
0017
0018                            *                 /* check end of block */
0019
0020 005014 B3C8                          CMPA.L   A0,A1            ;A1 - A0
0021 005016 62F8                          BHI      CMP_LOOP         ;if A1 > A0 then CMP_LOOP
0022 005018 B010                          CMP.B    (A0),D0          ;cmpare last
0023 00501A                     FOUND
0025 00501A 7000                          MOVEQ    #0,D0
0026 00501C 4E40                          TRAP     #0
0027
0028                            *                 ---------
0029                            *                 data area
0030                            *                 ---------
0031
0032 00501E =0000FFFF           ST_ADR    DS.B     $FFFF
0033 01501D 5A                  END_ADR   DC.B     $5A
0034
0035                                      END
```

7 2つのメモリブロックの内容がすべて一致しているか否かをチェックする

◉サンプルプログラム ［BCMP］

ファイルの内容を比較する場合などが，典型的な用途です．

■動作

メモリ～メモリ間の専用比較命令である　CMPM　(An)+,(An)+命令を使い，BLK__A とBLK__Bをそれぞれ先頭とする内容を比較する（ブロックの内容はバイトデータが256個としている）．

■各行の意味

行11～13：必要なレジスタを初期化している．

行14～17：比較ループ部で，両者が一致しないと，この時点でESC__CMPへ制御を移行する．

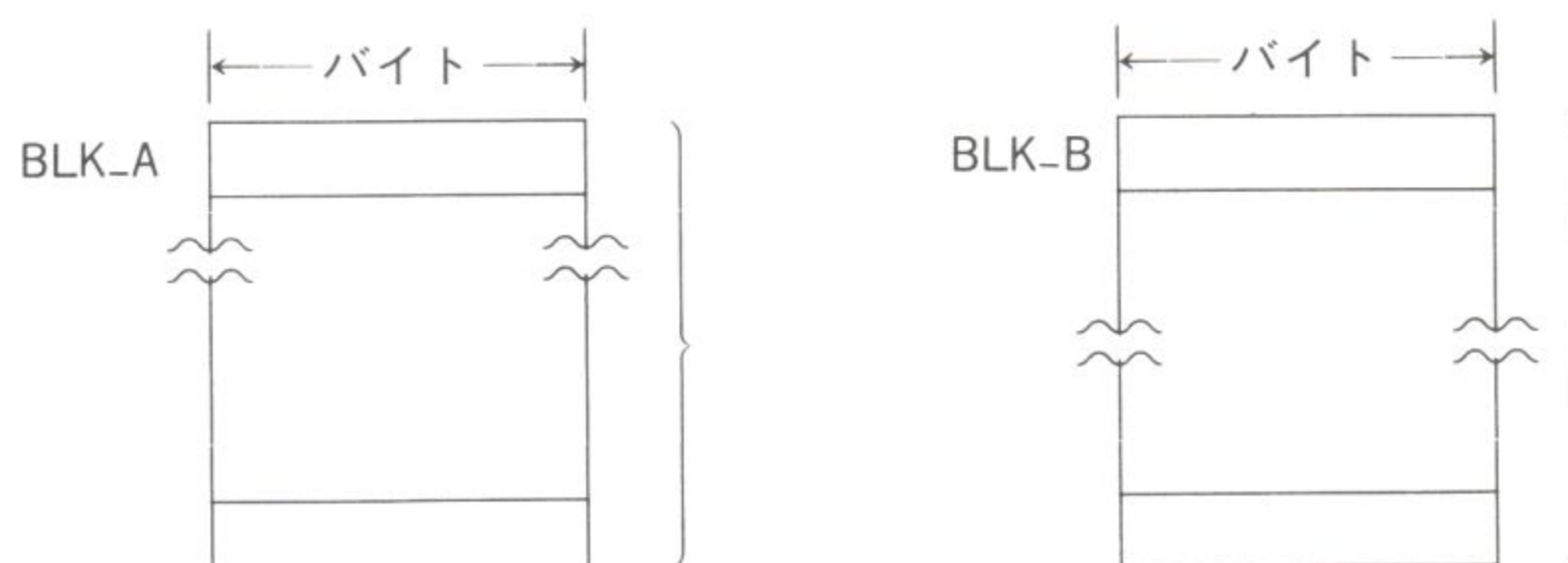
行27～28：BLK__A,BLK__Bを先頭とするロケーションから，256バイトの領域を予約している．

リスト［BCMP］

```
LINE ADDR.  CODE/VALUE          LABEL     OP        OPERAND          COMMENT

0005
0006         =00000100          BLK_LEN   EQU       256
0007
0009         =00005000                    ORG       $5000
0010
0011 005000 303C 00FF                     MOVE.W    #BLK_LEN-1,D0    ;set compare size to D0
0012 005004 41F9 0000 501C                LEA       BLK_A,A0         ;address pointer of BLK_A0
0013 00500A 43F9 0000 511C                LEA       BLK_B,A1         ;address pointer of BLK_A1
0014 005010                     CMP_LOOP
0015 005010 B308                          CMPM.B    (A0)+,(A1)+
0016 005012 6604                          BNE       ESC_CMP
0017 005014 51C8 FFFA                     DBRA      D0,CMP_LOOP
0018 005018                     ESC_CMP
0020 005018 7000                          MOVEQ     #0,D0
0021 00501A 4E40                          TRAP      #0
0022
0023                            *                  ---------
0024                            *                  data area
0025                            *                  ---------
0026
0027 00501C =00000100           BLK_A     DS.B      BLK_LEN
0028 00511C =00000100           BLK_B     DS.B      BLK_LEN
0029
0030                                      END
```

アプリケーション・ヒント

本例では不一致を検出した時点で作業を終了したが，一致しなかった件数をカウントしたい場合もある．

図2.13　BCMPの様子

両メモリブロックが同じものであるかをチェックする．

8 メモリ上に確保した64ビット値を符号反転する

◉サンプルプログラム ［NEGX］

NEG命令のサンプルであり，64ビット値の符号を反転するものです．

たとえば，対象となる値が1（イチ）なら－1（マイナス1）となるわけですが，最初に正符号であればそのまま，負であった場合には符号反転すれば，絶対値を求めたことになります．

■動作

DATAを先頭とする64ビットのメモリレジスタに1（イチ）を格納しておき，これを反転した内容（－1)をDATE_1からのメモリレジスタに，DATA_2のメモリレジスタには，同様にして－1を反転した内容（1）を格納するが，3つのメモリレジスタのサイズは，いずれも64ビットで構成される．

■要点

① NEG,NEGXによる演算はロングワード（4バイト，32ビット）で行う．
② まず下位32ビット，次に上位32ビットの順で行う．

■各行の意味

行 9～11：必要なレジスタを初期化している．

行13～16：DATAにあらかじめ用意してある1を－1に変換し，DATA_1へ格納する．
16行の命令を実行後，A0は9行で初期化した値にもどる．

行18～21：DATAの内容である－1を1に変換し，DATA_2へ格納する．

APPENDIX◉正の整数と負の整数の表現と絶対値

符号付き表現とビットパターンについてふれておきますが，大切なことは，要求されるアプリケーションを満足すればよいわけで，あくまでも，マシンそのものは2進数として処理しますから，符号はプログラマの解釈に委ねられるということです（ビットパターンを見やすくするため，4ビット（ニブルという）で区切っている）．

表2.4

ビットパターン	符号なし値	符号付き値	
0000 0000	0	0	
0000 0001	1	1	
・・・・・	・・・	・・・	
0111 1111	127	127	← $2^{n-1}-1$
1000 0000	128	－128	n＝8（8ビット）
1000 0001	129	－127	
・・・・・	・・・	・・・	
1111 1110	254	－2	
1111 1111	255	－1	← 255＝$2^{n}-1$ n＝8（8ビット）

note: 符号を考慮しない値を絶対値，符号付きを考慮した値を補数値ともいうが，通常の絶対値とは，
±1の絶対値は　1　である．
とした方が都合がよく，符号付（符号を考慮した上での）絶対値と呼ばれる．

アプリケーション・ヒント

先に述べたように，絶対値を求めるには，最上位の符号ビットが1（イチ）ならNEG命令を実行し，正符号へ変換する．

符号ビットが正なら符号変換操作は不要である（もし不安であれば，様々な値でテストしてみるとよい）．

図2.14 NEGXの様子

← ロングワード →

DATA	0	0	0	0	0	0	0	0	64ビットデータ（初期値1）
	0	0	0	0	0	0	0	1	
DATA_1	F	F	F	F	F	F	F	F	64ビットデータ（−1）
	F	F	F	F	F	F	F	F	
DATA_2	0	0	0	0	0	0	0	0	64ビットデータ（1）
	0	0	0	0	0	0	0	1	

リスト［NEGX］

```
LINE ADDR. CODE/VALUE          LABEL     OP       OPERAND        COMMENT

0005
0007        =00005000                  ORG      $5000
0008
0009 005000 41F9 0000 502E             LEA      DATA+8,A0      ;ini. pointer
0010 005006 43F9 0000 502E             LEA      DATA_1,A1
0011 00500C 45F9 0000 5036             LEA      DATA_2,A2
0012
0013 005012 44A0                       NEG.L    -(A0)
0014 005014 40A0                       NEGX.L   -(A0)
0015 005016 22D8                       MOVE.L   (A0)+,(A1)+    ;transter DATA to DATA_1
0016 005018 22D8                       MOVE.L   (A0)+,(A1)+
0017
0018 00501A 44A0                       NEG.L    -(A0)
0019 00501C 40A0                       NEGX.L   -(A0)
0020 00501E 24D8                       MOVE.L   (A0)+,(A2)+    ;transfer DATA to DATA_2
0021 005020 24D8                       MOVE.L   (A0)+,(A2)+
0022 005022                  BREAK
0024 005022 7000                       MOVEQ    #0,D0
0025 005024 4E40                       TRAP     #0
0026
0027                         *                 ---------
0028                         *                 data area
0029                         *                 ---------
0030
0031 005026 0000 0000        DATA      DC.L     0              ;upper 32 bit
0032 00502A 0000 0001                  DC.L     1              ;lower 32 bit
0033
0034 00502E =00000008        DATA_1    DS.L     2
0035 005036 =00000008        DATA_2    DS.L     2
0036
0037                                   END
```

7 論理演算命令サンプルプログラム

1 複数ビットを同時にクリアしたりビット列の取り出しを行う

◉サンプルプログラム ［AND］

特定のビットをクリアするには，BCLR命令が用意されていますが，AND命令を使用すると，任意のビットを同時にクリアしたり，特定のビット列を取り出すことができます．

■動作

① ワードデータのビット11，10，6，5，1の5ビットをクリアし，その他のビットを取り出す．

② CCRのビット4，3，2，1，0（X，N，Z，V，Cに対応）をクリアする．

■要点

ソースオペランドのビットパターンを，**表2.5**のように作成する．

表2.5

保存したい（取り出したい）ビット番号	“1”
無条件にクリアしたいビット番号	“0”

このようにすれば，ディスティネーション・オペランドの値にフィルタをかけることができる．

■各行の意味

行18〜19：19行のAND命令によって，D0に格納されている内容に対してフィルタをかけ，ビット番号11，10，6，5，1を無条件にクリアし，その他のビットを〝そっくり〟取り出している．

行21　：CCRのビット4〜0（X，N，Z，V，Cに対応）を無条件にクリアしている．

アプリケーション・ヒント

32ビットあれば32種類の情報を格納できるわけで，複数の条件を同時に満たしている項目を選び出せ，という場合に威力を発揮する（32回もビットテストや比較命令を実行しなくてよい）．

図2.15　ANDの様子

ANDI.W ＃MASK__A，D0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
MASK__A	1	1	1	1	0	0	1	1	1	0	0	1	1	1	0	1
AND)																
結果：D0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	0	0

←● MASK__Aの対応ビットが0なら，そのビットは無条件にクリア(0)される．
● MASK__Aの対応ビットが1なら，そのビットはD0の対応ビットをそのまま反映する．

ANDの真理値表

入力		出力
0	0	0
0	1	0
1	0	0
1	1	1

リスト[AND]

```
LINE ADDR.  CODE/VALUE         LABEL    OP       OPERAND            COMMENT

0005
0006                           *                  7654321076543210
0007        =0000F39D          MASK_A   EQU      %1111001110011101
0008
0009                           *                  765XNZVC
0010        =000000E0          MASK_B   EQU      %11100000
0011
0012                           *                  7654321076543210
0013        =0000AAAA          TEST     EQU      %1010101010101010
0014
0016        =00005000                   ORG      $5000
0017
0018 005000 303C AAAA                   MOVE.W   #TEST,D0
0019 005004 0240 F39D                   ANDI.W   #MASK_A,D0
0020
0021 005008 023C 00E0                   ANDI     #MASK_B,CCR
0022
0023 00500C 4E71                        NOP
0024
0025                                    END
```

NOTE　論理演算は，「データに対するフィルタ」と考えてもよく，要求される処理に応じた演算操作をすることで，都合のよい（必要な）データだけを取り出すことができる．

2 複数ビットを同時にセットしたりビット列の合成を行う

◉サンプルプログラム [OR]

特定のビットをセットするには，BSET命令が用意されていますが，OR命令を使用すると，任意のビットを同時にセットしたり，ビット列の合成をすることができます．

■動作

① ワードデータの下位バイトを保存しながら上位バイトのすべてのビットをセットする．

② D0とD1との合成を行う．

■要点

ソース・オペランドのビットパターンを，以下のように作成する．

表2.6

保存したい（取り出したい）ビット番号	"0"
無条件にセットしたいビット番号	"1"

このようにすれば，ディスティネーション・オペランドの値にフィルタをかけることができる．

■各行の意味

行24〜25：D0の内容の上位バイトは無条件にセットされ，下位バイトはそのまま保存される．

行27〜29：ここでは，D1とD2のビットのうち，"1"の立っているビットが合成される（この様子は24〜25と同じである）．

行31　：CCRのZビットだけをセットしている．

アプリケーション・ヒント

OR論理演算は，いずれかが"1"であれば無条件に"1"が得られるので，両者の条件のうち少なくとも一方が満たされているものを選択せよ，という場合に，複数の条件を1個の数値で表現できる．

図2.16　ORの様子

OR.B D1，D2

	7	6	5	4	3	2	1	0
D2	1	1	0	0	1	1	0	0
D1	0	0	1	1	0	0	1	1
OR)								
結果：D2	1	1	1	1	1	1	1	1

←1の立っているビットが合成される．

D1のビットが1なら無条件に対応ビットは1．

D1のビットが0ならD2の対応ビットがそのまま結果に出力される．

ORの真理値表

入力		出力
0	0	0
0	1	1
1	0	1
1	1	1

ORI.W #P_SET，D0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
P_SET	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
OR)																
結果：D0	1	1	1	1	1	1	1	1	0	1	0	1	0	1	0	1

リスト[OR]

```
LINE ADDR.  CODE/VALUE         LABEL    OP       OPERAND          COMMENT

0005
0006                           *                  7654321076543210
0007            =00005555      DATA     EQU      %0101010101010101
0008
0009                           *                  7654321076543210
0010            =0000FF00      P_SET    EQU      %1111111100000000
0011
0012                           *                  76543210
0013            =00000033      SYM_A    EQU      %00110011
0014
0015                           *                  76543210
0016            =000000CC      SYM_B    EQU      %11001100
0017
0018                           *                  765XNZVC
0019            =00000004      SET_Z    EQU      %00000100
0020
0022            =00005000               ORG      $5000
0023
0024 005000 303C 5555                   MOVE.W   #DATA,D0
0025 005004 0040 FF00                   ORI.W    #P_SET,D0
0026
0027 005008 123C 0033                   MOVE.B   #SYM_A,D1
0028 00500C 143C 00CC                   MOVE.B   #SYM_B,D2
0029 005010 8401                        OR.B     D1,D2
0030
0031 005012 003C 0004                   ORI      #SET_Z,CCR
0032
0033 005016 4E71                        NOP
0034
0035                                    END
```

3 複数ビットを同時に反転する

◉サンプルプログラム　[EOR]

特定のビットを反転するには，BCHG命令が用意されていますが，EOR命令を使用すると，任意のビットを同時に反転することができます．

■動作

① バイトデータの下位4ビットを保存しながら上位4ビットのすべてを反転する．
② CCRのXビットは保存しつつ，N，Z，V，Cビットを反転する．

■要点

ソース・オペランドのビットパターンを，以下のように作成する．

表2.7

保存したいビット番号	"0"
無条件に反転したいビット番号	"1"

このようにすれば，ディスティネーション・オペランドの値にフィルタをかけることができる．

■各行の意味

行18〜19：D0のビット7〜4を反転するが，ビット3〜0は保存する．

行21　　：CCRのN，Z，V，C，を反転するが，他のビットは保存する．

アプリケーション・ヒント

EOR論理演算命令は対称性をもっているので，データを暗号化したり，暗号化したデータを復元することにも応用できる．

図2.17　EORの様子

EORI.B ＃EOR_DATA, D0

	7	6	5	4	3	2	1	0
D0	0	1	0	1	0	1	0	1
EOR_DATA	1	1	1	1	0	0	0	0
EOR)								
結果：D0	1	0	1	0	0	1	0	1

EOR_DATAで0のパターンを指定しているので，結果はD0の対応ビットがそのまま得られる．

EOR_DATAで1のパターンを指定しているので，結果はD0の対応ビットを反転したパターンとなっている．

EORの真理値表

入力		出力
0	0	0
0	1	1
1	0	1
1	1	0

リスト[EOR]

```
LINE ADDR.  CODE/VALUE          LABEL     OP       OPERAND          COMMENT

0005
0006                            *                   76543210
0007             =00000055      DATA      EQU      %01010101
0008
0009                            *                   76543210
0010             =000000F0      EOR_DATA  EQU      %11110000
0011
0012                            *                   765XNZVC
0013             =0000000F      CHG_DATA  EQU      %00001111
0014
0016             =00005000                ORG      $5000
0017
0018 005000 103C 0055                     MOVE.B   #DATA,D0
0019 005004 0A00 00F0                     EORI.B   #EOR_DATA,D0
0020
0021 005008 0A3C 000F                     EORI     #CHG_DATA,CCR
0022
0023 00500C 4E71                          NOP
0024
0025                                      END
```

4 ビットパターンのすべてを反転する

◉サンプルプログラム ［NOT］

ビットパターンのすべてを反転する命令の用途も様々ですが，たとえば，メモリテストのテストパターンとしては，〝10″の連続したビットパターンと〝01″の連続したビットパターンが必要で，双方のデータを用意しなくとも，NOT命令で対処できます．

■動作

ロングワード値である1（イチ）を論理反転するが，ついでに，この値に1を加えると符号付き表現での-1（マイナス1）を得ることができる．

■各行の意味

行11〜12：1の補数をとる部分であり，＄00000001なら$FFFFFFFEとなる．

行13　：2の補数をとる部分であり，元のデータが$FFFFFFFEなら，$FFFFFFFFとなり，これは符号付き表現での-1（マイナス1）を意味する．

アプリケーション・ヒント

外部とインターフェースする場合，反転したデータを出力したり入力したりするケースも多く存在する．

図2.18　NOTの様子

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

NOT命令を実行すると，これらのすべてのビットパターンは反転するので，
＄00000001　は　＄FFFFFFFEとなる．

リスト［NOT］

```
LINE ADDR. CODE/VALUE           LABEL    OP        OPERAND        COMMENT

0005
0006         =00000001          DATA     EQU       1
0007
0009         =00005000                   ORG       $5000
0010
0011 005000 203C 0000 0001               MOVE.L    #DATA,D0
0012 005006 4680                         NOT.L     D0
0013 005008 5280                         ADDQ.L    #1,D0
0014
0015 00500A 4E71                         NOP
0016
0017                                     END
```

APPENDIX◉1の補数と2の補数

すべてのビットパターンを反転することを，「1の補数をとる」と言い，1の補数に1を加えることを，「2の補数をとる」と言う．2の補数とは，符号を反転したものを意味し，1なら-1（マイナス1）を得ることである．

8
シフト・ローテート命令サンプルプログラム

1 多倍精度のシフト

◉サンプルプログラム［SFT］

ビットパターンを左右に桁移動するという命令には，しばしばお世話になるもので，整数演算のみならず，パリティチェックやデータのエンコード／デコードなどにも応用できます．

■動作

① D0-D1-D2の32ビットレジスタを連結し，D0を最上位32ビット，D1を中位32ビット，D2を最下位32ビット，と考え，この内容を左へ1ビット論理シフトする．

② D3-D4-D5の32ビットレジスタを連結し，同様に右へ1ビット論理シフトする．

■要点

① 左への論理シフト命令はLSL，右へのそれはLSRであるが，これだけでは多倍精度の桁移動は不可能で，拡張ビットであるCCRのXビットを介した桁移動命令であるROXLやROXRを併用する．

② ROXL，ROXRでのXビットに関しては，命令実行前と実行後では意味が異なる．

表2.8　Xビットの意味

オペランドへ入力されるXビット	本命令によって押し出されたXビットではなく，他の命令によって押し出されたものである．
オペランドから出力されるXビット	本命令によって押し出されたものである．

■各行の意味

行11～13：D0-D1-D2にテストパターンをセットしている．

行15　：最下位32ビットを左へ1ビット桁移動するが，最上位ビットは外へ押し出されてXビットへ保持される．

行16　：D1の内容は左へ1ビット桁移動するが，最下位ビットはXビットで置き換わり，押し出された最上位ビットはXビットへ保持される．

行17　：D0に対して16行と同じ操作が行われるが，D0の最上位ビットの内容はCとXビットへ保持される（16行でもXビットだけでなくCビットにも最上位ビットが保持される）．

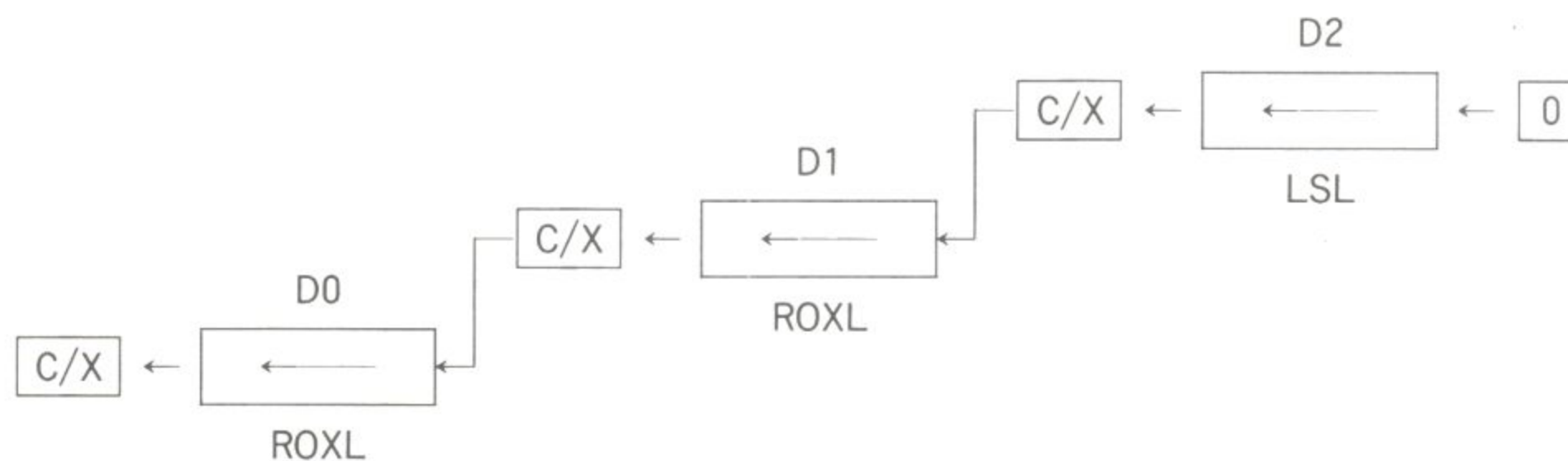
行19～25：桁移動方向が異なるが，内容は11～17行と同様である．

アプリケーション・ヒント

メモリ上に確保したメモリレジスタの内容に対しても同様な考え方で対処できるが，ここでのポイントも，やはりXビットということになる．もちろん，メモリアドレスの更新方法も重要である．

産業用分野の典型的な用途として，8ビットの入力ポートへ8種類の機器のステータスがインターフェースされ，各機器からのサービス要求の合図を〝1″とか〝0″の論理で決めてあることを想定する．

ビット8がONになれば，ビット8へインターフェースされた機器へのサービスを行うことができるが，複数のサービス要求の処理には，いずれかの方向へ桁移動し，XビットがONになった時点でサービスを行えばよい．また，優先順を管理するビット列を別に用意しておき，このビット列を回転することによって，優先順を割り当てることも可能である．

図2.19　SFTの様子

リスト[SFT]

```
LINE ADDR. CODE/VALUE          LABEL    OP        OPERAND          COMMENT

0005
0006         =AAAAAAAA         DATA     EQU       $AAAAAAAA
0007
0009         =00005000                  ORG       $5000
0010
0011 005000 203C AAAA AAAA              MOVE.L    #DATA,D0         ;D0-D1-D2
0012 005006 2200                        MOVE.L    D0,D1
0013 005008 2400                        MOVE.L    D0,D2
0014
0015 00500A E38A                        LSL.L     #1,D2
0016 00500C E391                        ROXL.L    #1,D1
0017 00500E E390                        ROXL.L    #1,D0
0018 005010                    BREAK_1
0019 005010 263C AAAA AAAA              MOVE.L    #DATA,D3         ;D3-D4-D5
0020 005016 2803                        MOVE.L    D3,D4
0021 005018 2A03                        MOVE.L    D3,D5
0022
0023 00501A E28B                        LSR.L     #1,D3
0024 00501C E294                        ROXR.L    #1,D4
0025 00501E E295                        ROXR.L    #1,D5
0026 005020                    BREAK_2
0027 005020 4E71                        NOP
0028
0029                                    END
```

APPENDIX◉左シフトと右シフト

n倍する：n回の左シフトによって，2のn乗倍を計算したことになるので，D0の内容を左へ3ビットだけシフトすれば，D0の内容は8倍される．

nで割る：n回の右シフトによって，2のn乗で割ったことになるので，D0の内容を右へ3ビットだけシフトすれば，D0の内容は8で割った商になる．

2 ワードデータの上位バイトと下位バイトを交換する

◉サンプルプログラム ［BSWAP］

32ビットデータの上位ワードと下位ワードとを入れ替えるSWAP命令はあっても，ビット15～0に対応するワードデータのSWAP命令はなく，ローテート（回転）命令のサンプルとして取り上げます．

■動作

オペランドを8回ローテートして，上位バイトと下位バイトを交換する．ただし，上位16ビットは保存する．

■要点

ROL，RORはXビットの影響を受けないので，単に8回だけ回転命令を繰り返せば目的を達成できる．

■各行の意味

行13～23：特に問題となる箇所はないはずであり，結果は同じであるが，ROLとRORで確認してみた．

アプリケーション・ヒント

特別にプログラマが用意した識別ビットを回転すれば，優先順位の管理に応用できる．SWAP命令と併用すれば，さらに複雑なスワップが可能である．

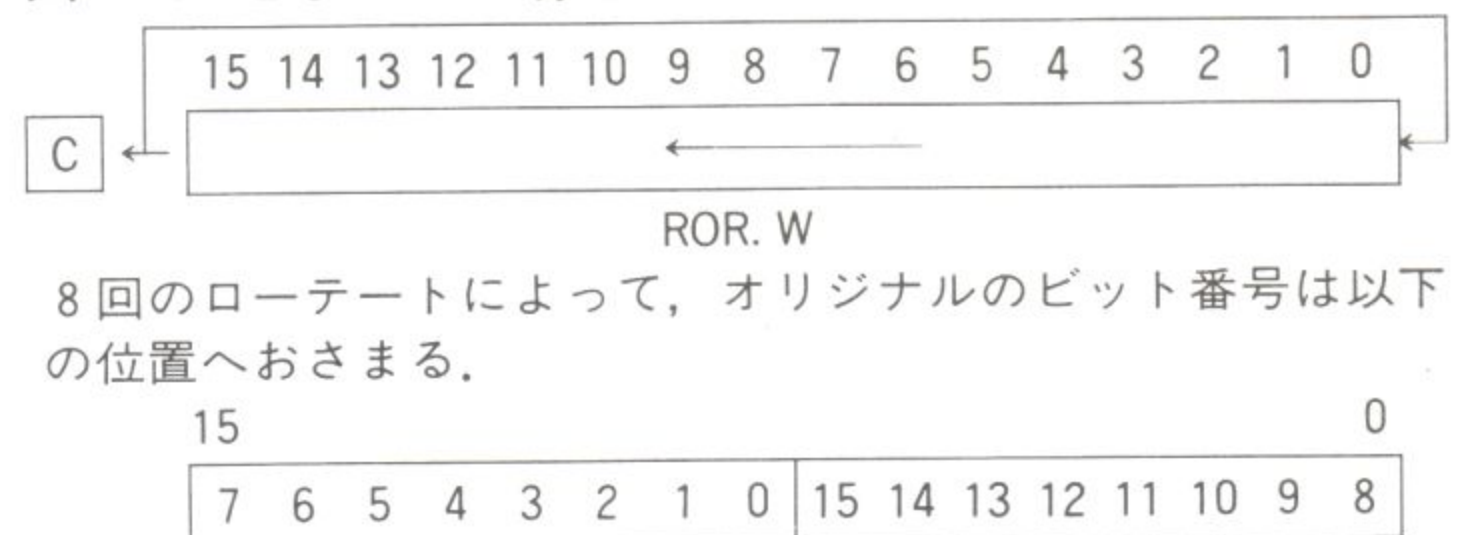
図2.20 BSWAPの様子

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

C ← [←]

ROR. W

8回のローテートによって，オリジナルのビット番号は以下の位置へおさまる．

15															0
7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8

リスト［BSWAP］

```
LINE ADDR. CODE/VALUE         LABEL     OP        OPERAND          COMMENT

0005
0006          =000012AB       DATA_0    EQU       $12AB
0007          =000012AB
0008          =000034CD       DATA_1    EQU       $34CD
0009
0011          =00005000                 ORG       $5000
0012
0013 005000 303C 12AB                   MOVE.W    #DATA_0,D0
0014 005004 E058                        ROR.W     #8,D0
0015
0016 005006 323C 12AB                   MOVE.W    #DATA_0,D1
0017 00500A E159                        ROL.W     #8,D1
0018
0019 00500C 343C 34CD                   MOVE.W    #DATA_1,D2
0020 005010 E05A                        ROR.W     #8,D2
0021
0022 005012 363C 34CD                   MOVE.W    #DATA_1,D3
0023 005016 E15B                        ROL.W     #8,D3
0024
0025 005018 4E71                        NOP
0026
0027                                    END
```

9
ビット演算命令・BCD演算命令サンプルプログラム

1　偶数値をカウントする

◉サンプルプログラム　[BIT]

ローテート／シフト命令と同様，ビット演算命令はアセンブラの得意とする繊細な記述をサポートする典型的命令であり，本例では単にビット0をテストしているだけですが，情報をビット単位で操作するための命令です．

■動作

メモリ上に確保した256バイトの領域に格納されているバイトデータのビット0をテストし，偶数データの数をカウントする．

■要点

① 偶数か奇数かはビット0の内容で判別でき，ゼロ（0）なら偶数，イチ（1）なら奇数である．

② テストしたいデータは連続したロケーションに配置されているので，(An)＋によるアドレッシングモードを適用するが，ビット演算命令のデータサイズは，オペランドがメモリ上に存在する場合には，無条件にバイトサイズと解釈されてしまうことを思い出してほしい．

③ テスト条件によってD1をカウントアップするか否かを決定するが，どのケースで分岐させたらよいのか，つまりBNEなのかBEQなのかということも重要な選択である．

■各行の意味

行11〜13：A0（アドレスポインタ），D0（テストする回数），D1（偶数データの個数を格納する），を初期化している．

行15〜17：BNEのNEはNot Equalのことで，テストの結果，そのビットが〝1〞であり，〝0〞に等しくなかったことを意味する．つまり，奇数であるからカウントアップは不要であるので，NEXTというアドレスへ分岐している．

アプリケーション・ヒント

16行の条件式を変更すれば，奇数データの個数をカウントできるのは明らかである．

ビットごとに「ある情報」を記憶させたり，記憶させた情報を否定したり反転できるので，情報が広範囲に拡散することはなく，プログラマが管理しやすくなる．

図2.21 BITの様子

リスト[BIT]

```
LINE ADDR. CODE/VALUE         LABEL    OP       OPERAND         COMMENT

0005
0006        =00000100         LEN      EQU      256
0007
0009        =00005000                  ORG      $5000
0010
0011 005000 41F9 0000 501A             LEA      MEM,A0
0012 005006 303C 00FF                  MOVE.W   #256-1,D0
0013 00500A 4281                       CLR.L    D1
0014 00500C                   LOOP
0015 00500C 0818 0000                  BTST     #0,(A0)+
0016 005010 6602                       BNE      NEXT
0017 005012 5281                       ADDQ.L   #1,D1
0018 005014                   NEXT
0019 005014 51C8 FFF6                  DBRA     D0,LOOP
0020
0021 005018 4E71                       NOP
0022
0023                          *                ---------
0024                          *                data area
0025                          *                ---------
0026
0027 00501A =00000100         MEM      DS.B     LEN
0028
0029                                   END
```

2 BCD10桁の加算を行う

◉サンプルプログラム ［ABCD］

アセンブラの摘要範囲では，BCD表現が要求されることは少なく，データ構造面的にはバイナリ型よりも効率が低下します．しかし，私たちの慣れている10進数を扱えるというメリットがありますし，桁数も容易に拡張できます．

■動作

10桁のBCD数を格納できるメモリレジスタを2つ（BCD__A，BCD__B）確保し，

(BCD__A)＋(BCD__B)

を計算し，結果をBCD__Bへ求める．

■要点

① メモリレジスタ間の加算であるから，-(An)によるアドレッシングを適用する．
② 桁数は10桁（5バイト）であるので，アドレスレジスタは，メモリレジスタの先頭から5番地進んだ地点に初期化する．16桁ならメモリレジスタのサイズは8であるので，先頭から8番地進んだ地点となる．
③ 演算の直前でCCRのXビットをクリア，Zビットをセットする．

■各行の意味

行12〜15：必要なレジスタを初期化している．
CCRへの転送は演算の直前でなければならない．

行16〜18：BCDデータの加算部分．

行26〜27：10桁のテストデータをメモリレジスタへ初期化しているが，`$′マークを付加し，16進で定義している．
どのようにメモリへ格納されるかは，左のオブジェクトコード・フィールドを御覧いただきたい．
もし単に12と記述すれば，アセンブラは10進の「十二」と解釈し，メモリへは$0Cとして格納してしまう．同様に，99「九十九」と記述すると，$63として格納してしまう．

デバッガで結果を確認してみると，BCD__Bの内容は，

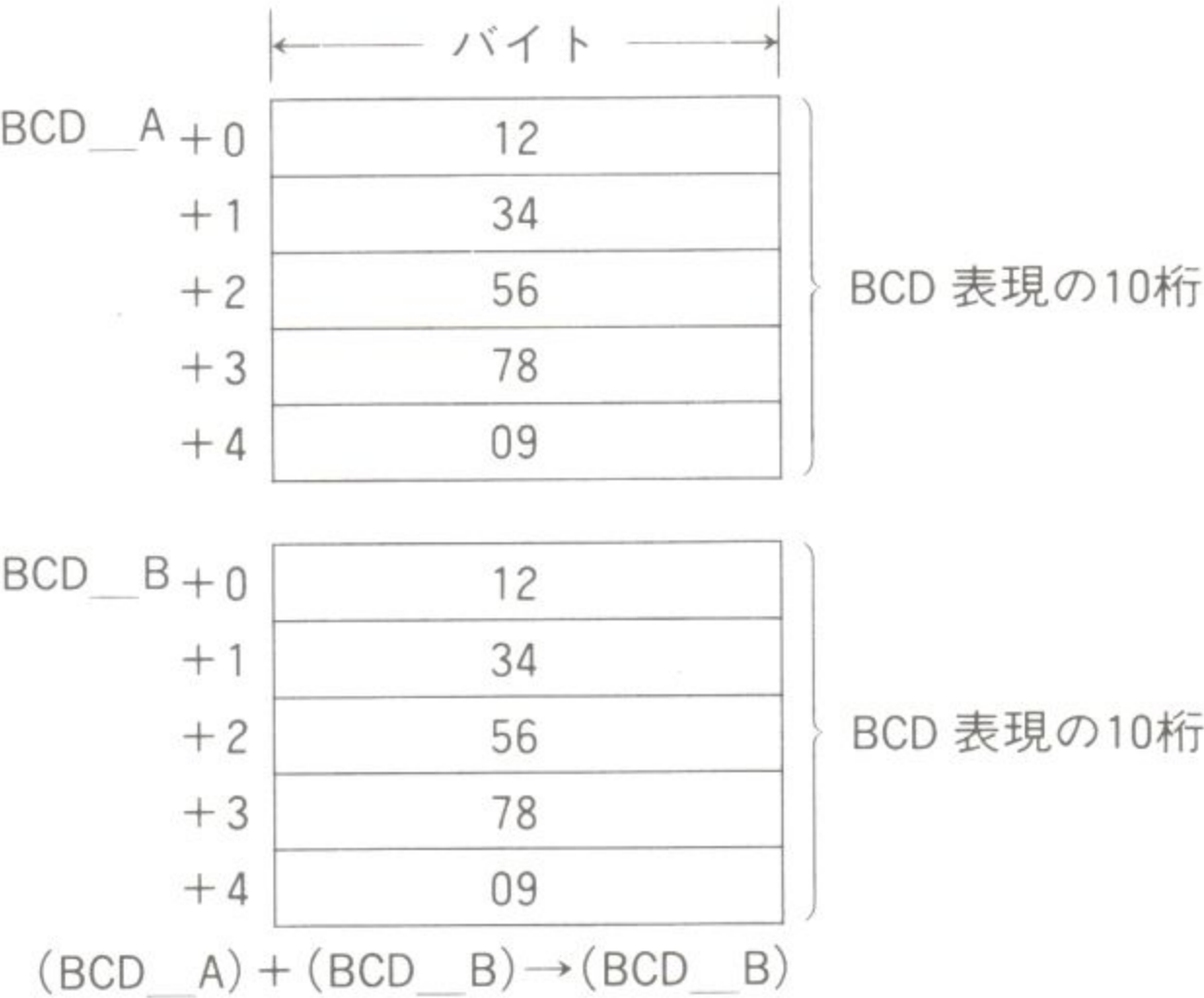
＄24，＄69，＄13，＄56，＄18

が上位桁から格納され，ABCD命令によって，2469135618という10桁のBCD表現の値が得られることがわかる．

アプリケーション・ヒント

BCD表現の減算命令であるSBCDに関しても，使い方は同様である．

NBCDは符号を考慮したBCD演算に使用される命令で，BCD表現の値の補数をとるものである．

図2.22 ABCDの様子

APPENDIX◉BCD(2進化10進数)

コンピュータにとってBCDは特別な表現であるので，BCD専用の演算命令が用意され，加算ならADDとABCDのように，それぞれ別々の約束で加算が実行され，結果も前者はバイナリ表現（0000～1111までのビットパターン），後者はBCD表現（0000～1001までのビットパターン）となる．

BCDコードのビットパターンは0000～1001までが使用され，1010～1111は演算の対象外となる．このようにして0～9までの10進数を表現し，16進表記での`A´～`F´の部分は存在しない．

1バイトは8ビットであるから，2桁のBCD数を格納できるが，10進の00～99の範囲しかカバーできない．バイナリ表現では，0～255の範囲をカバーできるので，コンピュータにとっては，あまり効率のよい表現方法ではないことがわかる．

符号表現であるが，1010～1111までの6種類のパターンが余るので，使用するコンピュータの命令セットを考慮し，アルゴリズムを決定し，適当な符号パターンを選択する（詳細は演算のアルゴリズムについて解説された文献などを御覧いただきたい）．

表2.9
数値表現とビットパターン

数値表現	コンピュータの内部表現（ビットパターン）
0	0 0 0 0
1	0 0 0 1
2	0 0 1 0
3	0 0 1 1
4	0 1 0 0
5	0 1 0 1
6	0 1 1 0
7	0 1 1 1
8	1 0 0 0
9	1 0 0 1

リスト「ABCD」

```
LINE ADDR.  CODE/VALUE          LABEL     OP      OPERAND          COMMENT

0005
0006                            *                  765XNZVC
0007          =00000004         Z_CLR     EQU     %00000100
0008
0010          =00005000                   ORG     $5000
0011
0012 005000 41F9 0000 501F                LEA     BCD_A+5,A0
0013 005006 43F9 0000 5024                LEA     BCD_B+5,A1
0014 00500C 7004                          MOVEQ   #5-1,D0          ;ini. loop counter
0015 00500E 44FC 0004                     MOVE    #Z_CLR,CCR       ;reset X-bit,set Z-bit
0016 005012                     ADD_BCD
0017 005012 C308                          ABCD    -(A0),-(A1)      ;(BCD_A) + (BCD_B) -> (BCD_B)
0018 005014 51C8 FFFC                     DBRA    D0,ADD_BCD
0019
0020 005018 4E71                          NOP
0021
0022                            *                 -------------
0023                            *                 BCD data area
0024                            *                 -------------
0025
0026 00501A 1234 5678 09        BCD_A     DC.B    $12,$34,$56,$78,$09
0027 00501F 1234 5678 09        BCD_B     DC.B    $12,$34,$56,$78,$09
0028
0029                                      END
```

10 無条件分岐命令とジャンプ・テーブル

ジャンプ・テーブルは，指定されたメニュー番号に従った処理へ制御を移行する場合に利用され，あらかじめそれぞれの処理ルーチンをメモリのどこかに予約しておき，メニュー番号をキーにして処理アドレスを取り出し，所定の処理へ分岐させます．

ここでの要点は次の２点である．

① あらかじめ処理先を定義したジャンプ・テーブルを作成すること．

② 条件を０からスタートする数値に変換（コード化）し，これをインデックスとしてジャンプ・テーブルを参照すること．

1 D0の内容をキーにしてそれぞれの処理へ分岐させる

◉サンプルプログラム ［JMP］

■動作

処理数に制限はないが，ここではＤ０にセットされた０～３までの値をキーにして，４通りの処理へ分岐させている．

Ｄ０と処理先との関係は**表2.10**のように対応し，動作の確認はメッセージをスクリーンへ表示することで行っている．

表2.10
D0と処理先の関係

D0の内容	処理先のラベル
0	PRG__0
1	PRG__1
2	PRG__2
3	PRG__3

■解法

４つの処理ルーチンへ分岐させるための命令を順に予約しておき，Ｄ０をインデックスにして，その命令が記述されているアドレスへ分岐させればよい．

■ジャンプ・テーブルの作成

図2.23 ジャンプ・テーブル

J__TBL＋0	JMP PRG__0という命令を格納しておく
	NOP命令を格納
＋8	JMP PRG__1という命令を格納しておく
	NOP命令を格納
＋16	JMP PRG__2という命令を格納しておく
	NOP命令を格納
＋24	JMP PRG__3という命令を格納しておく
	NOP命令を格納

■ジャンプ・テーブルの参照

JMP命令の置かれているアドレスはJ__TBLから0，８，16，24，のように８番地間隔であるから，ベース・アドレスにD0を８倍した値を加算したアドレスへ分岐させればよい．

ここでNOPという意味のない命令がJMP命令の後に置かれているが，テーブル・サイズは6より8のほうが計算しやすいからである．

ジャンプ先アドレスは，d16(An,IX)のアドレッシングを適用すればよいから，

A0にベース・アドレス（J__TBL）
D0にキー（キーを8倍したインデックス値）

を設定し，

```
JMP 0(A0,D0.L)
```

により，目的の分岐命令の置かれているアドレスへジャンプさせればよい．

■各行の意味

行9～11：ベース・アドレスからD0の値を8倍した地点へ分岐している．

行13～28：メッセージ出力部分で，PRG 0へ制御が移行すると，〝PRG 0〟とスクリーンへ表示する．

行24～25：本システムでは，A0に文字列先頭アドレスをセットしてこのようにすると，スクリーンへ文字列が表示される．ただし文字列は〝$〟記号で終了する約束になっている．

行27～28：本システムでは，このようにするとシステムへもどることになっている．

行33～40：各処理ルーチンへの分岐命令が記述されているジャンプ・テーブルが位置するが，テーブル・サイズを8バイトとするためにJMP命令とNOPで1組とした．

行46～49：各処理へエントリしたかどうかを確認するためのメッセージを格納してある．

APPENDIX◉間接アドレッシングと分岐命令について

分岐命令での間接アドレッシングはアドレス表現そのものであり，たとえば，A0に＄5000がセットされている状態で以下の命令を実行すれば，

```
JMP (A0)
```

＄5000番地へ分岐する．

これを，＄5000番地から格納されているアドレスを取り出し，そのアドレスへ分岐する，と解釈しないでいただきたい（この様子は他のメモリ間接アドレス・モードにも適用されるので，心配なら本リストでテストされるとよいでしょう）．

リスト[JMP]

```
XA68K  Rev.2.1  SOURCE LIST                                        PAGE: 001

LINE ADDR. CODE/VALUE         LABEL    OP       OPERAND         COMMENT

0005
0007        =00005000                  ORG      $5000
0008
0009 005000 41F9 0000 5040             LEA      J_TBL,A0        ;load base address to A0
0010 005006 E788                       LSL.L    #3,D0           ;D0=D0*8
0011 005008 4EF0 0800                  JMP      0(A0,D0.L)
0012
0013 00500C 41F9 0000 5060    PRG_0    LEA      MSG_0,A0
0014 005012 4EF9 0000 5036             JMP      MSG_OUT
0015
0016 005018 41F9 0000 5068    PRG_1    LEA      MSG_1,A0
0017 00501E 4EF9 0000 5036             JMP      MSG_OUT
0018
0019 005024 41F9 0000 5070    PRG_2    LEA      MSG_2,A0
0020 00502A 4EF9 0000 5036             JMP      MSG_OUT
0021 005030
0022 005030 41F9 0000 5078    PRG_3    LEA      MSG_3,A0
0023 005036                   MSG_OUT
0024 005036 103C 0009                  MOVE.B   #9,D0
0025 00503A 4E40                       TRAP     #0
0027 00503C 7000                       MOVEQ    #0,D0
0028 00503E 4E40                       TRAP     #0
0029
0030                          *                ----------
0031                          *                jump table
0032                          *                ----------
0033 005040 4EF9 0000 500C    J_TBL    JMP      PRG_0
0034 005046 4E71                       NOP
0035 005048 4EF9 0000 5018             JMP      PRG_1
0036 00504E 4E71                       NOP
0037 005050 4EF9 0000 5024             JMP      PRG_2
0038 005056 4E71                       NOP
0039 005058 4EF9 0000 5030             JMP      PRG_3
0040 00505E 4E71                       NOP
0041
0042                          *                -----------
0043                          *                string area
0044                          *                -----------
0045
0046 005060 5052 4720 300D    MSG_0    DC.B     "PRG 0",$0D,$0A,'$'
            0A24
0047 005068 5052 4720 310D    MSG_1    DC.B     "PRG 1",$0D,$0A,'$'
            0A24
0048 005070 5052 4720 320D    MSG_2    DC.B     "PRG 2",$0D,$0A,'$'
            0A24
0049 005078 5052 4720 330D    MSG_3    DC.B     "PRG 3",$0D,$0A,'$'
            0A24
0050
0051                                   END
```

2 D0の内容をキーにして処理アドレスを取り出しキーに応じた処理先へ分岐させる

◉サンプルプログラム ［JTBL］

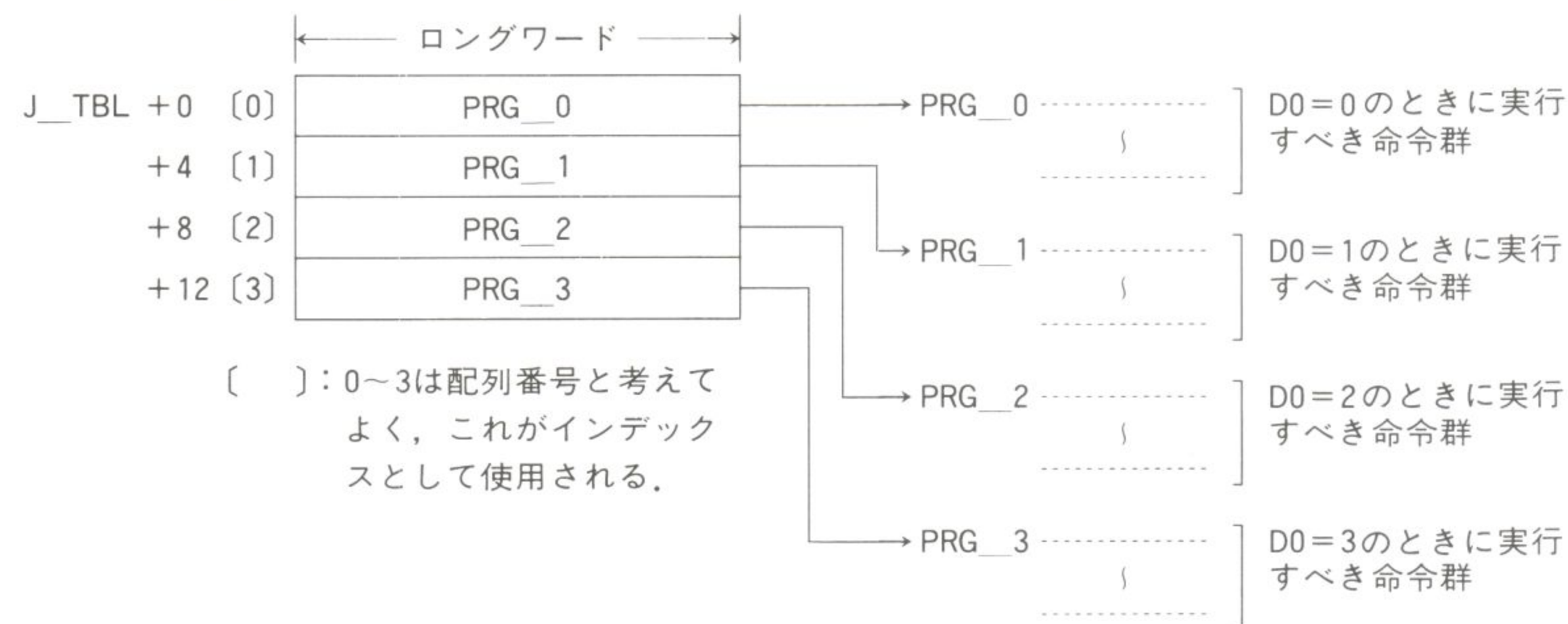
■動作

処理数に制限はないが，ここではD0にセットされた0～3までの値をキーにして，4通りの処理へ分岐させている．先の例よりこちらの方が応用度が高いので，アセンブラによるプログラミングには不可欠である（動作の確認はメッセージを出力することで行った）．

■解法

メモリ上に処理先のアドレス表（ジャンプ・テーブル）を作成しておき，D0をインデックスにして表を参照（処理アドレスを取り出す）し，指定された処理へ分岐する．

■ジャンプ・テーブルの作成

図2.24 JTBLの様子

■処理アドレスの参照

アドレスは4バイト（ロングワード）なので，目的の処理アドレスが格納されている表の位置は，インデックスを4倍した値だけ離れていることになる．

そこでD0の内容が3なら，J__TBLから12だけ進んだアドレスに格納されている処理アドレスを取り出して分岐すればよい．

■各行の意味

行9～11：D0をインデックスにしてジャンプ・テーブルから処理アドレスをA0に取出す．

行12　：A0で指定されるアドレスへ分岐する．
もしA0の内容が$501Aなら$501A番地へ分岐し,A0でポイントされるメモリから分岐アドレスを取り出して分岐するのではない．

行14～23：PRG 0～PRG 3とは各処理ルーチンのエントリであり，本来ならこれ以後に，そのための処理プログラムが長々と記述されているはずである．

行34～37：ジャンプ・テーブルが位置し，その内容は先の例のように命令ではなく処理先のアドレス（エントリ・アドレス）である．

リスト［JTBL］

```
LINE ADDR. CODE/VALUE         LABEL    OP       OPERAND          COMMENT

0005
0007        =00005000                  ORG      $5000
0008
0009 005000 41F9 0000 5042             LEA      J_TBL,A0         ;load base address to A0
0010 005006 E588                       LSL.L    #2,D0            ;D0=D0*4
0011 005008 2070 0800                  MOVEA.L  0(A0,D0.L),A0    ;pickup entry address to A0
0012 00500C 4ED0                       JMP      (A0)             ;jump
0013
0014 00500E 41F9 0000 5052    PRG_0    LEA      MSG_0,A0
0015 005014 4EF9 0000 5038             JMP      MSG_OUT
0016
0017 00501A 41F9 0000 505A    PRG_1    LEA      MSG_1,A0
0018 005020 4EF9 0000 5038             JMP      MSG_OUT
0019
0020 005026 41F9 0000 5062    PRG_2    LEA      MSG_2,A0
0021 00502C 4EF9 0000 5038             JMP      MSG_OUT
0022 005032
0023 005032 41F9 0000 506A    PRG_3    LEA      MSG_3,A0
0024 005038                   MSG_OUT
0025 005038 103C 0009                  MOVE.B   #9,D0            ;message out then return
0026 00503C 4E40                       TRAP     #0
0028 00503E 7000                       MOVEQ    #0,D0
0029 005040 4E40                       TRAP     #0
0030
0031                          *                ----------
0032                          *                jump table
0033                          *                ----------
0034 005042 0000 500E         J_TBL    DC.L     PRG_0
0035 005046 0000 501A                  DC.L     PRG_1
0036 00504A 0000 5026                  DC.L     PRG_2
0037 00504E 0000 5032                  DC.L     PRG_3
0038
0039                          *                -----------
0040                          *                string area
0041                          *                -----------
0042
0043 005052 5052 4720 300D    MSG_0    DC.B     "PRG 0",$0d,$0a,'$'
            0A24
0044 00505A 5052 4720 310D    MSG_1    DC.B     "PRG 1",$0d,$0a,'$'
            0A24
0045 005062 5052 4720 320D    MSG_2    DC.B     "PRG 2",$0d,$0a,'$'
            0A24
0046 00506A 5052 4720 330D    MSG_3    DC.B     "PRG 3",$0d,$0a,'$'
            0A24
0047
0048                                   END
```

3 1文字コマンドによる処理ルーチンへの分岐

◉サンプルプログラム ［MENU］

これまでの例では，インデックスがコード化されているものとしましたが，ここではコマンドとして入力される1文字をインデックスとしてコード化する過程を扱っています.

■動作

Y，U，Z，W，S，D，の各コマンドに応じた処理へ分岐させるもので，コマンドと処理ルーチンとは以下のように対応している.

また処理先へ制御が移行したことを確認するために，"Y"が入力されると"Y command"というメッセージを表示させ，いずれにも該当しない場合は "Command Error" と表示する.

表2.11

コマンド	処理先
Y	Y_CMD
U	U_CMD
Z	Z_CMD
W	W_CMD
S	S_CMD
D	D_CMD

■解法

分岐のアルゴリズムはこれまでの例と同様であるが，Y～Dを一連の文字列と考え，入力された文字がこれらの文字列の何番目に位置しているかを求め，これをインデックスとして分岐先のアドレスを取り出す.

■文字位置を求める

Y，U，Z，W，S，D，なる文字列に対して，Yなら0，Dなら5，を求めることができれば，あとはこれまでの例と同様にして各処理へ分岐できる.

そこでD0に "Y" という文字コードが格納されているとき，Y～Dまで比較ループを回し，見つかった場所から文字位置を求めるが，ループ・カウンタ値は文字位置を意味する値に使えるので，サーチする文字列をD～Yのように逆にしている.

いずれにしても "Y" が入力されれば0，"D" なら5を求める.

■各行の意味

行15～16：D1にコマンド文字列より1少ない値をセットしループ値とする.A1にはコマンド文字列の先頭アドレスをセットする.

行17～20：D0にセットされているコマンド・コードとコマンド文字列とを比較し，D0の内容が何番目に位置しているかをD1に求める.D0の内容が文字列中に存在すればFOUNDへ分岐する.

行21～23：指定コマンドが存在しなかった場合はここへ制御が移行し，エラー・メッセージを表示してシステムへもどる.

行26～30：D1にはインデックスが格納されているので，これから処理先のアドレスを取り出して分岐する.

行35～83：これまでの内容と同一である.

アプリケーション・ヒント

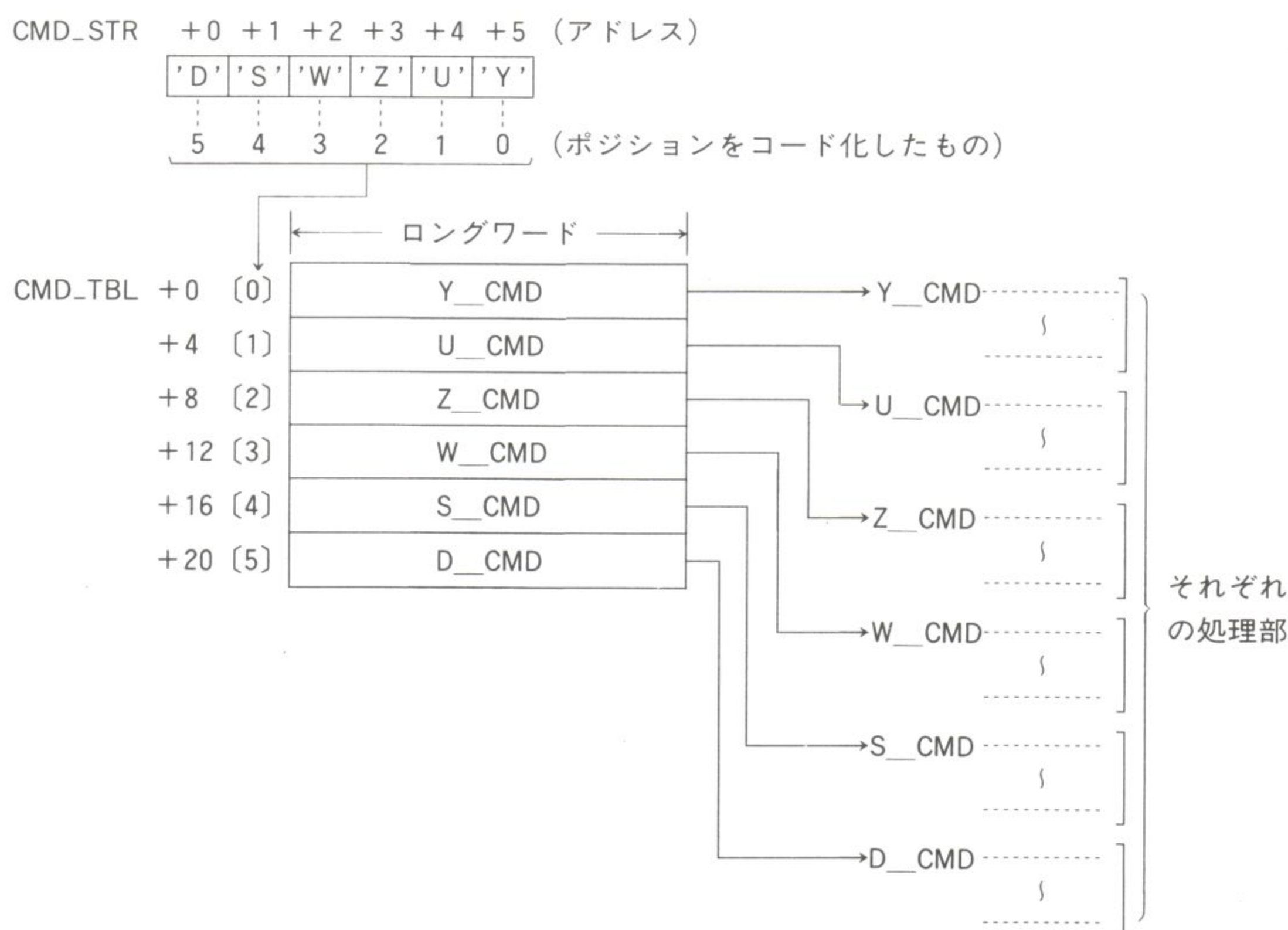
■インデックスと分岐テーブルの構成

ポイントはコマンド文字をどうインデックスへ変換するか，そのインデックスから参照できる処理先はどこか，であって，処理先アドレスが格納されたジャンプ・テーブルの順序を本例と逆にしてもよいし，コマンド文字を逆からサーチしてもよい．

■インデックスの計算方法

たとえばメニュー番号として`0´($30)～`9´($39)を予定していれば，$30を減じるだけで$0～$9の数値が得られ，即座にインデックスとして機能する．

あるいは一度得られたコードに簡単な算術演算や論理演算，またはこれらを組み合わせてコード化することもあれば，必要なアドレスや値を取り出すために，複数のテーブルを参照しながら処理を進めることもある．

図2.25
MENUの様子

リスト[MENU]

```
LINE ADDR.  CODE/VALUE          LABEL     OP       OPERAND          COMMENT

0005
0006         =00000006          CMD_LEN   EQU      6
0007
0009         =00005000                    ORG      $5000
0010
0011                            *                  --------------------
0012                            *                  search key position
0013                            *                  --------------------
0014         =00005000          MENU
0015 005000  223C 0000 0005               MOVE.L   #CMD_LEN-1,D1    ;D1: setup loop counter
0016 005006  43F9 0000 5092               LEA      CMD_STR,A1       ;A1: ini. address pointer
0017 00500C                     CMP_LOOP
0018 00500C  B019                         CMP.B    (A1)+,D0
0019 00500E  6710                         BEQ      FOUND
0020 005010  51C9 FFFA                    DBRA     D1,CMP_LOOP
0021 005014                     NOT_FOUND
0022 005014  41F9 0000 5098               LEA      MSG_ERR,A0       ;error message
```

```
0023 00501A 4EF9 0000 5070              JMP      MSG_OUT
0024
0025                          *                  jump to command
0026 005020                   FOUND
0027 005020 E589                        LSL.L    #2,D1          ;key=key*4
0028 005022 45F9 0000 507A              LEA      CMD_TBL,A2     ;A2: base address
0029 005028 2472 1000                   MOVE.L   0(A2,D1),A2
0030 00502C 4ED2                        JMP      (A2)
0031
0032                          *                  ----
0033                          *                  proc
0034                          *                  ----
0035 00502E                   Y_CMD
0036 00502E 41F9 0000 50A8              LEA      MSG_Y,A0       ;Y command
0037 005034 4EF9 0000 5070              JMP      MSG_OUT
0038 00503A                   U_CMD
0039 00503A 41F9 0000 50B4              LEA      MSG_U,A0       ;U command
0040 005040 4EF9 0000 5070              JMP      MSG_OUT
0041 005046                   Z_CMD
0042 005046 41F9 0000 50C0              LEA      MSG_Z,A0       ;Z command
0043 00504C 4EF9 0000 5070              JMP      MSG_OUT
0044 005052                   W_CMD
0045 005052 41F9 0000 50CC              LEA      MSG_W,A0       ;W command
0046 005058 4EF9 0000 5070              JMP      MSG_OUT
0047 00505E                   S_CMD
0048 00505E 41F9 0000 50D8              LEA      MSG_S,A0       ;S command
0049 005064 4EF9 0000 5070              JMP      MSG_OUT
0050 00506A                   D_CMD
0051 00506A 41F9 0000 50E4              LEA      MSG_D,A0       ;D command
0052 005070                   MSG_OUT
0053 005070 103C 0009                   MOVE.B   #9,D0          ;mesage out then return
0054 005074 4E40                        TRAP     #0
0056 005076 7000                        MOVEQ    #0,D0
0057 005078 4E40                        TRAP     #0
0058
0059                          *                  -------------
0060                          *                  command table
0061                          *                  -------------
0062 00507A 0000 502E         CMD_TBL   DC.L     Y_CMD
0063 00507E 0000 503A                   DC.L     U_CMD
0064 005082 0000 5046                   DC.L     Z_CMD
0065 005086 0000 5052                   DC.L     W_CMD
0066 00508A 0000 505E                   DC.L     S_CMD
0067 00508E 0000 506A                   DC.L     D_CMD
0068
0069                          *                  --------
0070                          *                  strings
0071                          *                  --------
0072 005092 4453 575A 5559    CMD_STR   DC.B     "DSWZUY"
0073
0074                          *                  ------------
0075                          *                  message area
0076                          *                  ------------
0077 005098 436F 6D6D 616E    MSG_ERR   DC.B     "Command Error",$0d,$0a,'$'
            6420 4572 726F
            720D 0A24
0078 0050A8 5920 436F 6D6D    MSG_Y     DC.B     "Y Command",$0d,$0a,'$'
            616E 640D 0A24
0079 0050B4 5520 436F 6D6D    MSG_U     DC.B     "U Command",$0d,$0a,'$'
            616E 640D 0A24
0080 0050C0 5A20 436F 6D6D    MSG_Z     DC.B     "Z Command",$0d,$0a,'$'
            616E 640D 0A24
0081 0050CC 5720 436F 6D6D    MSG_W     DC.B     "W Command",$0d,$0a,'$'
            616E 640D 0A24
0082 0050D8 5320 436F 6D6D    MSG_S     DC.B     "S Command",$0d,$0a,'$'
            616E 640D 0A24
0083 0050E4 4420 436F 6D6D    MSG_D     DC.B     "D Command",$0d,$0a,'$'
            616E 640D 0A24
0084
0085        =00005000                   END      MENU
```

11 条件分岐命令後の処理

ある条件をテストし，その結果に従ってプログラムの流れを要求される方向へ向けるには，条件分岐命令（Bcc）を使いますが，まず基本的な2つの問題点を整理しておきます．

■Bcc命令に関すること

① 分岐範囲は2バイトのオフセットである（68000の全域をカバーできないが，これで十分である）．

② ある2つの値の大小関係などがテスト条件の中心となるが，扱われる数が符号付というケースは非常に稀であり，〝符号なし〟のアセンブラ表現に慣れることが先決である．

表2.12　A,Bを符号なし整数としたときの大小関係

CMP A,B　;内部では(B)-(A)を実行			
関係式	表現(cc)	関係式	表現(cc)
B＝A	EQ	B≠A	NE
B＞A	HI	B≦A	LS
B＜A	CS	B≧A	CC

(cc)：Condition Code
EQ：Equal
HI：High
CS：Carry Set
NE：Not Equal
LS：Low or Same
CC：Carry Clear

③ 同じ処理をするにも条件式は2通り考えられる点にも注意が必要である．つまり「等しい場合に何かをする」と「等しくない場合に何もしない」は同じ表現になるので，どのような条件の場合に分岐させるのか，あるいは分岐させないのか，という選択も重要である．

■条件テスト後の処理には2つのタイプがある

TYPE_1：条件を満足した場合に何かを実行し，それ以外は何もしないタイプ．
TYPE_2：条件を満足した場合としなかった場合に，それぞれ実行するものがあるタイプ．

図2.26　TYP_1のフローチャート

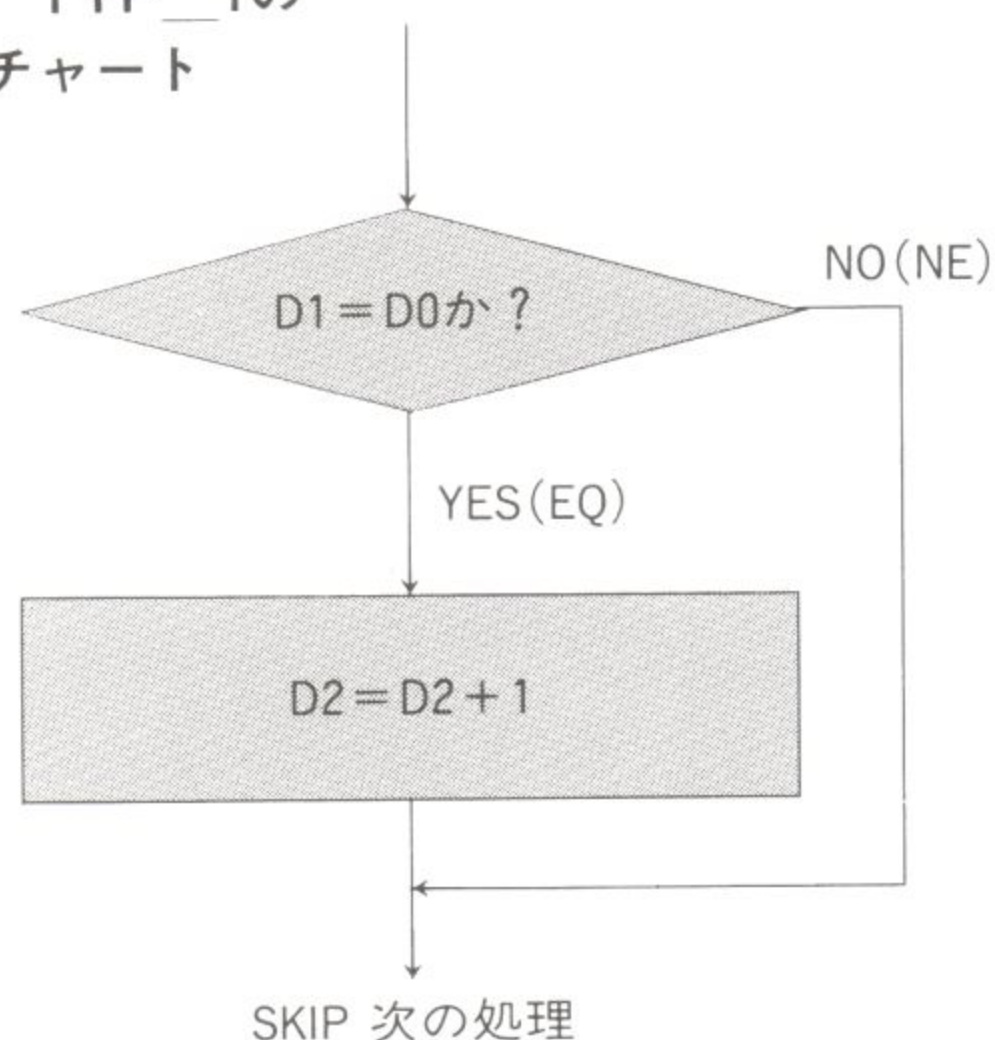
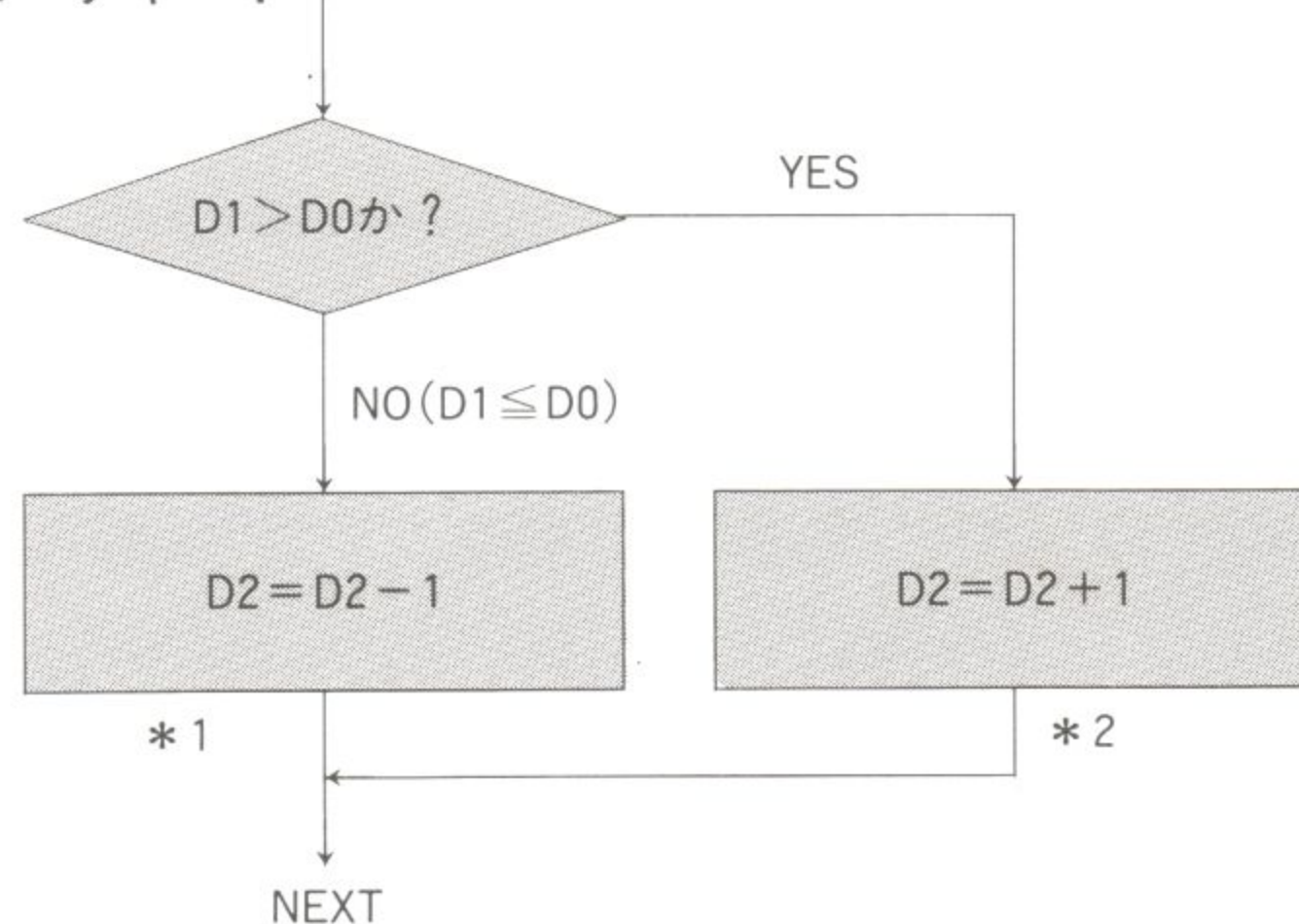
D1＝D0か？
NO(NE)
YES(EQ)
D2＝D2＋1
SKIP 次の処理

note：フローチャートによる表現では，YES/NOの位置関係はどのようにしてもよいし，本図のNOを左側に記述することもある．

図2.27　TYP_2のフローチャート

D1＞D0か？
YES
NO(D1≦D0)
D2＝D2－1
D2＝D2＋1
＊1
＊2
NEXT

note：＊1または＊2の後には，無条件分岐命令（BRA NEXT）を使用しないとNEXTへ移行できない．本例では＊1の後にBRA NEXTを置いている．

1 D0とD1を比較し等しい場合はD2をインクリメントする

◉サンプルプログラム ［TYP＿1］

タイプ1に分類される条件処理で，一方の条件を満たした場合に何かを実行し，そうでない場合は次へ制御を移行する例です．

■解法

条件は

EQ …… D2をインクリメント
NE …… 何もしない（次の処理へ分岐させる）

であるから，何もしないように条件NEで分岐させればよい．

リスト［TYP＿1］

```
LINE ADDR. CODE/VALUE        LABEL    OP       OPERAND        COMMENT

0005
0007         =00005000                ORG      $5000
0008         =00005000       TYP_1
0009 005000 B280                      CMP.L    D0,D1          ;(D1)-(D0)
0010 005002 6602                      BNE      SKIP
0011 005004 5282                      ADDQ.L   #1,D2          ;D2=D2+1
0012 005006                  SKIP
0014 005006 7000                      MOVEQ    #0,D0
0015 005008 4E40                      TRAP     #0
0016
0017         =00005000                END      TYP_1
```

2 D0とD1を比較しD1>D0ならD2をインクリメント，それ以外はD2をデクリメントする

◉サンプルプログラム ［TYP＿2］

タイプ2に分類される条件処理で，条件を満足した場合としない場合の両方に実行すべき処理がある．

■解法

条件，アセンブラ表現，処理，の関係は**表2.13**のようになる．

表2.13
条件・アセンブラ表現、処理の関係

条件	表現	処理内容
D1＞D0	HI	D2＝D2＋1
D1≦D0	LS	D2＝D2－1

インクリメントあるいはデクリメントの後,どこへ制御を移行するかという問題があり,いずれかの処理後,無条件分岐命令(BRA)を置かねばならない．要するに，インクリメントまたはデクリメント後の次が正規の位置ならよいが，その次にデクリメントあるいはインクリメントを実行する部分が位置していては困るわけだ．

アプリケーション・ヒント

処理内容によっては別々の方向へ制御が移行し，元の位置へもどらないこともある．以下のことに注意すべきである．

① 所定処理へ分岐させるための表現の選択．

② 分岐先には何を実行する命令を置けばよいか．

リスト[TYP＿2]

```
LINE ADDR. CODE/VALUE         LABEL    OP       OPERAND       COMMENT

0005
0007          =00005000                ORG      $5000
0008          =00005000       TYP_2
0009 005000 B280                       CMP.L    D0,D1         ;(D1)-(D0)
0010 005002 6204                       BHI      INC_D2
0011 005004                   DEC_D2
0012 005004 5382                       SUBQ.L   #1,D2         ;D2=D2-1
0013 005006 6002                       BRA      NEXT
0014 005008                   INC_D2
0015 005008 5282                       ADDQ.L   #1,D2         ;D2=D2+1
0016 00500A                   NEXT
0018 00500A 7000                       MOVEQ    #0,D0
0019 00500C 4E40                       TRAP     #0
0020
0021          =00005000                END      TYP_2
```

3 D0とD1を比較し等しいか否かをスクリーンへ表示する

●サンプルプログラム ［EQ］

分類上はタイプ2ですが，タイプ1的な処理を行うこともできます．

■解法

メッセージを表示するには，メッセージが格納されているアドレスをスイッチすればよいので，比較命令を行う直前で1方の条件を満たしておく．そして比較命令後の処理は，すでに満たされている場合は分岐し，それ以外なら改める．

ここでは以下のようにしている．

① まずD1＝D0であると仮定し，A0には〝等しい〟を表示する文字列の先頭アドレスを格納する．

② 比較の結果，先の仮定に反しなければ分岐し，それ以外は〝等しくない〟を表示する文字列の先頭アドレスをA0に格納する．

いずれにしても，処理は〝PRINT〟というラベルへ移行する．

アプリケーション・ヒント

ある条件に従って2通りの値を設定したい場合，条件テストを行う前に1方の条件を満足しておき，条件命令後に残りの条件を満たす．

このような処理も頻繁に要求され，〝1〟か〝0〟かで処理結果を表現したい場合，処理へ入る前に〝1〟にセットし，最後に残りの条件に応じた処理を行う．

リスト［EQ］

```
LINE ADDR. CODE/VALUE          LABEL    OP       OPERAND        COMMENT

0005
0007        =00005000                   ORG      $5000
0008
0009 005000 41F9 0000 501A              LEA      MSG_EQ,A0
0010 005006 B280                        CMP.L    D0,D1
0011 005008 6706                        BEQ      PRINT
0012 00500A 41F9 0000 5029              LEA      MSG_NE,A0
0013 005010                    PRINT
0014 005010 103C 0009                   MOVE.B   #9,D0
0015 005014 4E40                        TRAP     #0
0016
0018 005016 7000                        MOVEQ    #0,D0
0019 005018 4E40                        TRAP     #0
0020
0021                           *                --------
0022                           *                msg area
0023                           *                --------
0024
0025 00501A 4431 2045 5155     MSG_EQ   DC.B     "D1 EQUAL TO D0$"
            414C 2054 4F20
            4430 24
0026 005029 4431 204E 4F54     MSG_NE   DC.B     "D1 NOT EQUAL TO D0$"
            2045 5155 414C
            2054 4F20 4430
            24
0027
0028                                    END
```

12 ループ構造

ある命令群を繰り返し実行するプログラム構造をループ構造といい，ループ処理の終了条件によって，繰り返し指定タイプと終了条件指定タイプに分類できます．

■繰り返し回数指定タイプ

実行すべき回数があらかじめ決定している場合，ループの入り口で初期値を設定し，この値がゼロになったらループを脱出するが，実行回数と使用する命令を使い分ける必要がある．

① 実行回数が1～65536である場合は，繰り返し実行する命令の先頭をLOOPで表現するとき，以下のように整理できるが，実行回数0を要求するのであれば，条件指定タイプを採用すべきである．

表2.14

	Dnの初期値	実行回数	
DBRA Dn, LOOP （初期値＋1回実行される）	0	1	
	65534	65535	
	65535	65536	$FFFF：65535

② 実行回数が1～4294967296である場合

表2.15

	Dnの初期値	実行回数	
SUBQ.L #1, Dn BNE　LOOP （初期値＝0を除き初期値だけ実行される）	1	1	
	65535	65535	
	0	4294967296	$FFFFFFFF：4294967295

■終了条件指定タイプ

回数ではなく終了条件で制御されるので，極端な場合，1回も実行されないケースや無限に実行するケースなど，「何回実行するか」ではなく，「指定条件が満たされるまで」ループを脱出しない．

したがって，終了条件をまずチェックし，その結果に応じて繰り返すのか否かを決定する．

1 DBRA命令と繰り返し回数の設定

◉サンプルプログラム ［LOOP_1］

1回実行するとデータ・レジスタをインクリメント（＋1）し，ループ動作の確認をしています．1回多いとか少ないというミスは時として致命的なので，必ず本例のようなテストを実行しておくとよいでしょう．

■解法

DBRA Dn, LOOP

は，Dnをデクリメントし，$FFFF（-1）になると次の命令へ制御を移行し，それ以外はLOOPというラベルへ分岐するというものである．

Dnをループ・カウンタと呼ぶが，この値は必ずワードであるので，1バイトで十分であっても上位に＄00を満たす必要がある．

■各行の意味

行 9～13：D0の初期値が0の時，D1＝1を確認．

行15～19：D0の初期値が1の時，D2＝2を確認．

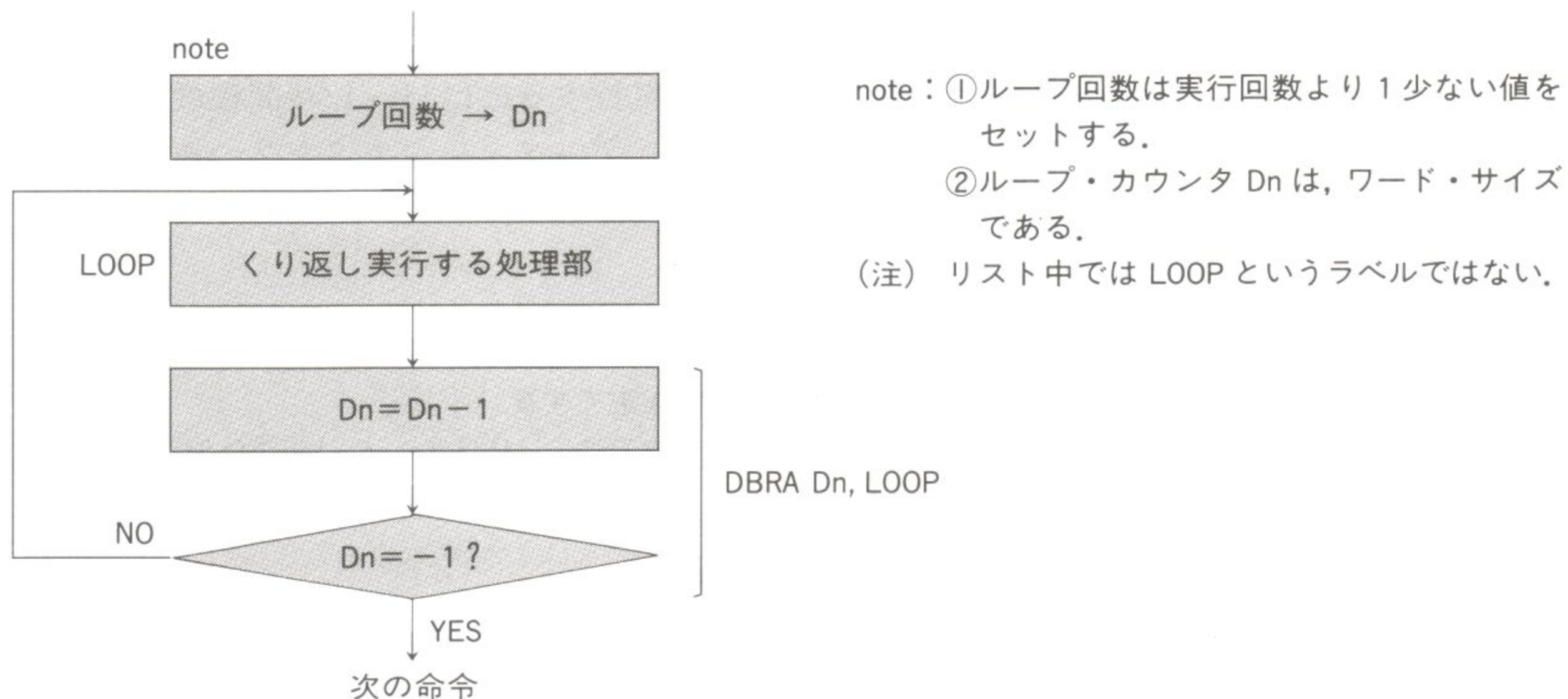
行21～25：D0の初期値が$FFFFの時，D3＝＄10000（65536）を確認．

アプリケーション・ヒント

① いずれの場合もループによって実行される命令はADDQ命令だけであるが，ラベルで指定されるループの先頭からDBRA命令の直前に位置する命令までに，かなりの命令が記述されることになるので，制御構造を明確にするために11，17，23行のようにラベルだけの行とすべきである．

② ループ・カウンタを初期化後，DBRAへ直接分岐させれば，実行回数はカウンタ値と同じになる．

③ 場合によっては「n回をm回実行する」というようなネスティングが要求されることもある．

図2.28 LOOP_1の様子（DBRAによるループ制御）

リスト[LOOP_1]

```
LINE ADDR.  CODE/VALUE        LABEL     OP        OPERAND       COMMENT

0005
0007         =00005000                  ORG       $5000
0008         =00005000        LOOP_1
0009 005000 7000                        MOVEQ.L   #0,D0         ;D0 = 0
0010 005002 4281                        CLR.L     D1
0011 005004                   COUNT_1
0012 005004 5281                        ADDQ.L    #1,D1         ;< countup D1 >
0013 005006 51C8 FFFC                   DBRA      D0,COUNT_1
0014 00500A
0015 00500A 7001                        MOVEQ.L   #1,D0         ;D0 = 1
0016 00500C 4282                        CLR.L     D2
0017 00500E                   COUNT_2
0018 00500E 5282                        ADDQ.L    #1,D2         ;< countup D2 >
0019 005010 51C8 FFFC                   DBRA      D0,COUNT_2
0020
0021 005014 70FF                        MOVEQ.L   #-1,D0        ;D0 = $FFFF
0022 005016 4283                        CLR.L     D3
0023 005018                   COUNT_3
0024 005018 5283                        ADDQ.L    #1,D3         ;< countup D3 >
0025 00501A 51C8 FFFC                   DBRA      D0,COUNT_3
0026
0028 00501E 7000                        MOVEQ     #0,D0
0029 005020 4E40                        TRAP      #0
0030 005022
0031         =00005000                  END       LOOP_1
```

2 DBRAを使用しないループ制御

◉サンプルプログラム ［LOOP_2］

手作業でループ・カウンタをデクリメントし，その後Bcc命令で分岐するか否かを決定する．

■解法

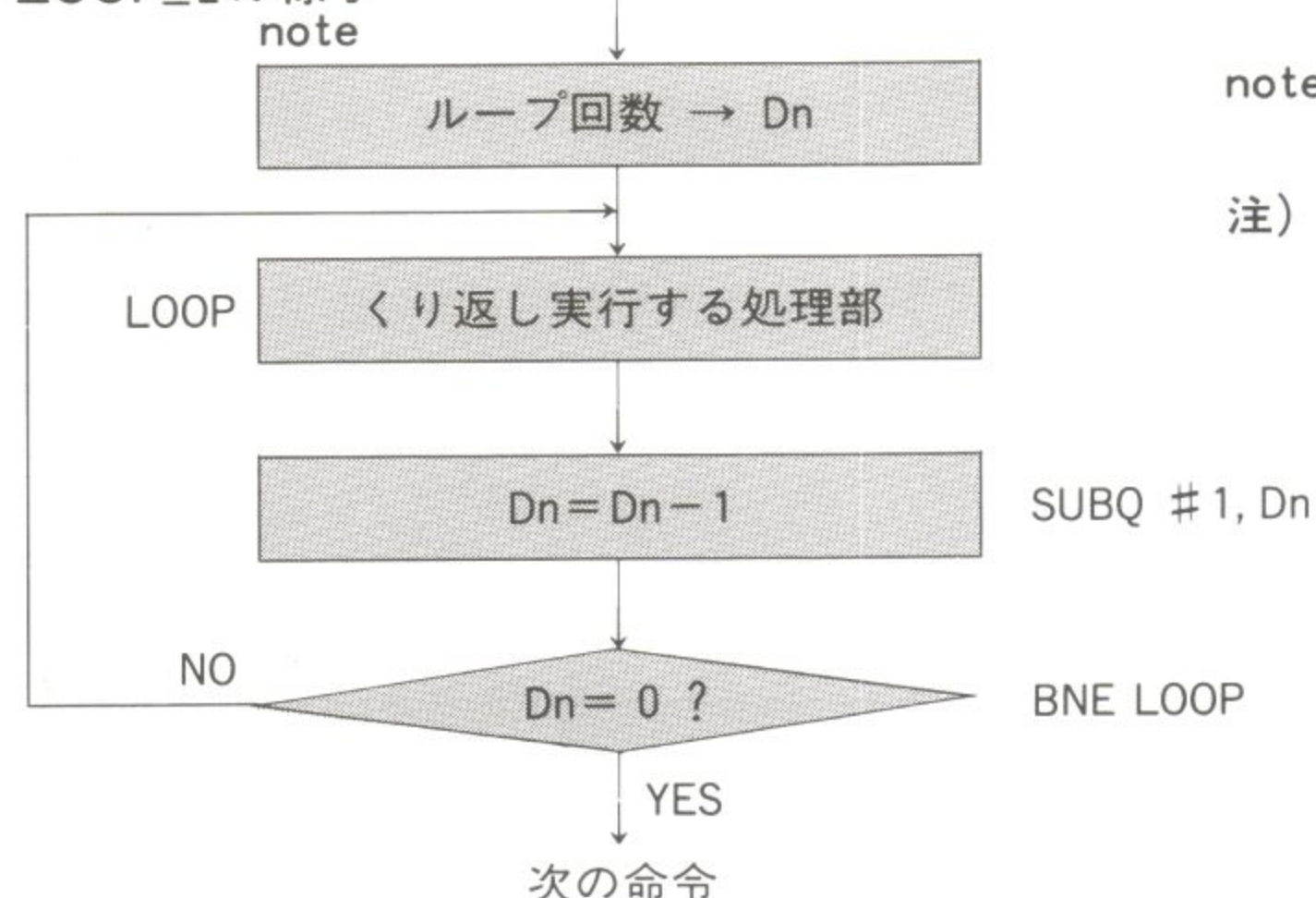
ループ・カウンタの初期値はループ回数と同一であるが，ここではカウンタをワードとし，65536回のインクリメントを実行するために初期値を0にしている．

所定処理を実行後SUBQでカウンタをデクリメントし，ゼロでないならCOUNT_1へ分岐し，それ以外はBNE命令の次に記述されている命令へ制御を移行する．

アプリケーション・ヒント

ループ・カウンタがワード・サイズならDBRAでよいが，ロングワードでは4294967296回までカウントできるので，8Mの68000でADDQをこれだけ実行するには11〜12時間程度必要である．これ以上のカウント値が要求されるのであれば，データ・レジスタを2つ連結して倍精度でのデクリメントを実行し，以後同様に処理すればよい．

図2.29
LOOP_2の様子

note：実行回数そのものをセットするが，ロングワード値まで指定できる．
注） リスト中では LOOP というラベルではない．

リスト［LOOP_2］

```
LINE ADDR. CODE/VALUE        LABEL     OP       OPERAND        COMMENT

0005
0007        =00005000                 ORG      $5000
0008        =00005000        LOOP_2
0009 005000 4280                      CLR.L    D0
0010 005002 4281                      CLR.L    D1
0011 005004                  COUNT_1
0012 005004 5281                      ADDQ.L   #1,D1          ;< countup D1 >
0013
0014 005006 5340                      SUBQ.W   #1,D0
0015 005008 66FA                      BNE      COUNT_1
0016 00500A                  BREAK
0018 00500A 7000                      MOVEQ    #0,D0
0019 00500C 4E40                      TRAP     #0
0020
0021        =00005000                 END      LOOP_2
```

3 多重ループ

◉サンプルプログラム ［LOOP_3］

DBRAを使用した3重ループの例ですが，68000は内部レジスタが豊富なので，多重ループもきわめて容易に構成できます．

■動作

内側から外へ向かって，それぞれ5，15，20回ループするので，全体では1500回実行することになる．

実行する命令はD1をインクリメントする命令であり，外側のループを脱出する時のD1は1500（5＊15＊20）となる．

■各行の意味

行13　　：D1をクリアし加算に備える．

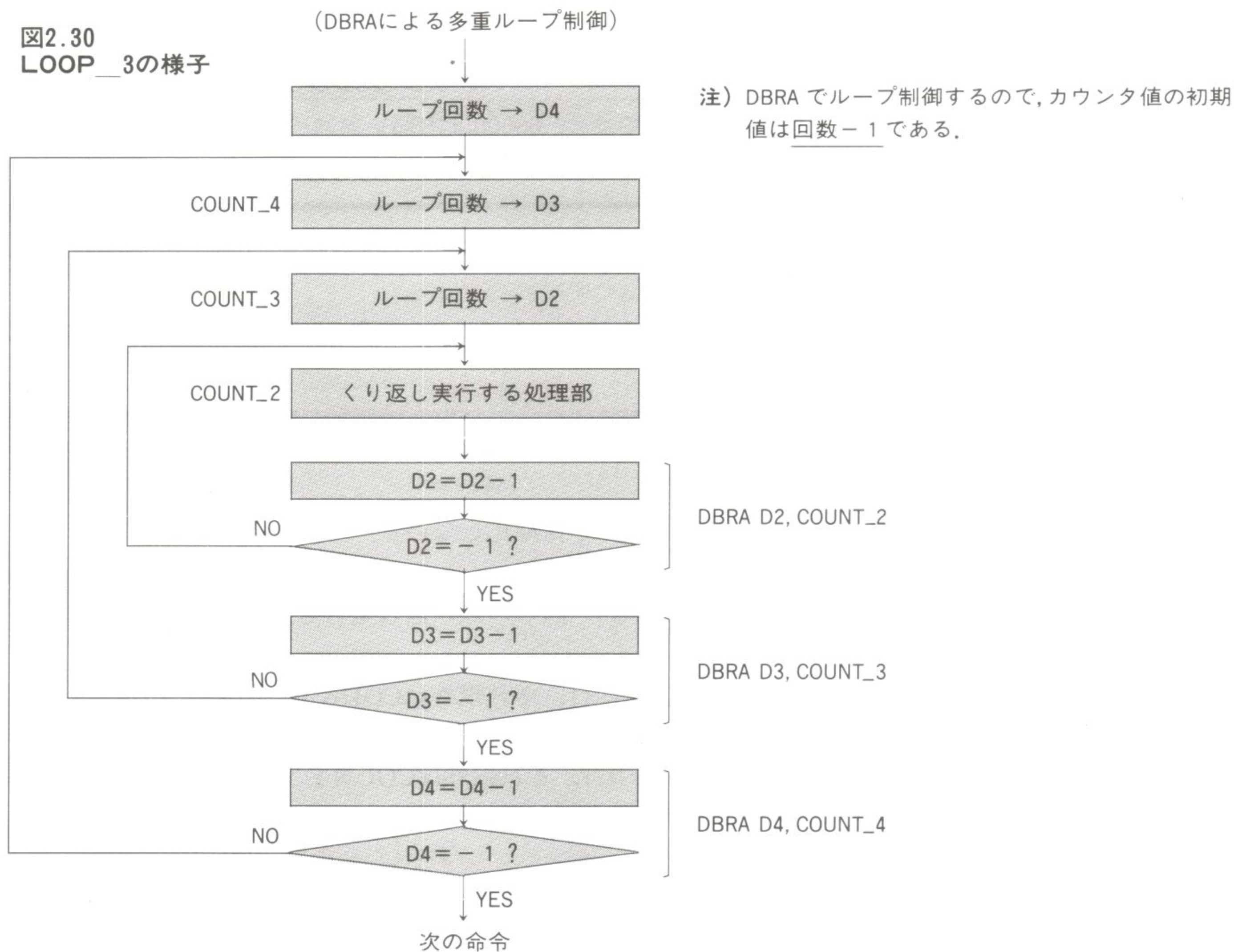
行15～19：それぞれのループ・カウンタを初期化している．

行21　　：この命令は1500（5＊15＊20）回だけ繰り返し実行される．

行22　　：D2が$FFFF（-1）なら次の行，それ以外はCOUNT 2へ分岐．

行23　　：D3が$FFFF（-1）なら次の行，それ以外はCOUNT 3へ分岐．COUNT 3へ制御が移行するたびに内側のループが実行される．

行24　　：D4が$FFFF（-1）なら次の行，それ以外はCOUNT 4へ分岐．COUNT 4へ制御が移行するたびに内側のループが実行される．

図2.30
LOOP__3の様子

リスト[LOOP_3]

```
LINE ADDR. CODE/VALUE         LABEL     OP       OPERAND          COMMENT

0005
0006        =00000005         LCOUNT_2 EQU      5
0007        =0000000F         LCOUNT_3 EQU      15
0008        =00000014         LCOUNT_4 EQU      20
0009
0011        =00005000                   ORG      $5000
0012        =00005000         LOOP_3
0013 005000 4281                        CLR.L    D1
0014
0015 005002 283C 0000 0013              MOVE.L   #LCOUNT_4-1,D4 ;ini. counter D4
0016 005008                   COUNT_4
0017 005008 263C 0000 000E              MOVE.L   #LCOUNT_3-1,D3 ;ini. counter D3
0018 00500E                   COUNT_3
0019 00500E 243C 0000 0004              MOVE.L   #LCOUNT_2-1,D2 ;ini. counter D2
0020 005014                   COUNT_2
0021 005014 5281                        ADDQ.L   #1,D1          ;D1 = D1+1
0022 005016 51CA FFFC                   DBRA     D2,COUNT_2
0023 00501A 51CB FFF2                   DBRA     D3,COUNT_3
0024 00501E 51CC FFE8                   DBRA     D4,COUNT_4
0025
0027 005022 7000                        MOVEQ    #0,D0
0028 005024 4E40                        TRAP     #0
0029
0030        =00005000                   END      LOOP_3
```

4 0(ゼロ)で終了する文字列をスクリーンへ出力する

◉サンプルプログラム ［WHILE_1］

同じ処理を繰り返し実行したいが，その回数が不明である場合の処理です．

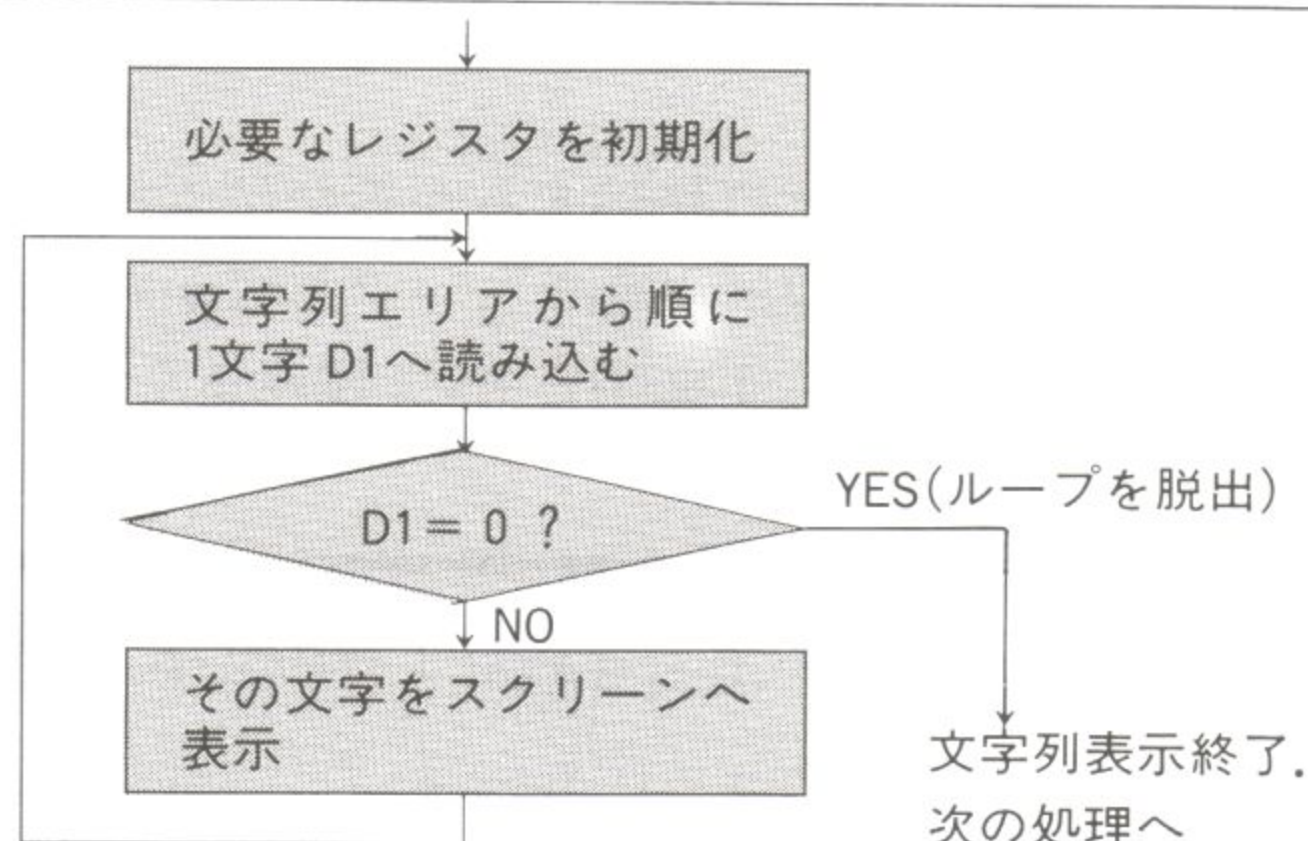
■解法

文字列領域から1文字読み込む際に使用するMOVE命令はどのようなデータを読み込んだのかをCCRのN，Zに反映するので，BEQでスクリーン出力を脱出すればよい．読み込んだ最初の1文字が＄00ならその場でループ処理を終了するので，1文字も表示されない．

■各行の意味

行9〜10：A0に文字列の先頭アドレス，D0にスクリーン出力の機能番号を設定．

行12〜15：＄00を読み込むまで1文字ずつスクリーンへ表示する（＄00は表示しない）．

図2.31 WHILE_1の様子

リスト［WHILE_1］

```
LINE ADDR. CODE/VALUE          LABEL    OP       OPERAND          COMMENT

0005
0007        =00005000                   ORG      $5000
0008        =00005000          WHILE_1
0009 005000 41F9 0000 5016              LEA      P_DATA,A0
0010 005006 103C 0002                   MOVE.B   #2,D0            ;function no.2 (print out)
0011 00500A                    P_OUT
0012 00500A 1218                        MOVE.B   (A0)+,D1
0013 00500C 6704                        BEQ      P_END
0014 00500E 4E40                        TRAP     #0
0015 005010 60F8                        BRA      P_OUT
0016 005012                    P_END
0018 005012 7000                        MOVEQ    #0,D0
0019 005014 4E40                        TRAP     #0
0020
0021                           *        -------------
0022                           *        printout data
0023                           *        -------------
0024 005016 0D0A 3D3D 2054     P_DATA   DC.B     $0d,$0a,"== Test data ==",$0d,$0a,0
            6573 7420 6461
            7461 203D 3D0D
            0A00
0025
0026        =00005000                   END      WHILE_1
```

5 バッファ領域への1行読み込み

◉サンプルプログラム ［WHILE_2］

終了条件が2つある例です.

■動作

キーボードから1文字ずつ最高で80文字読み込むが，C/R（キャリッジ・リターン）コードが入力されれば終了する.

■解法

終了条件は次の2つである.

① C/Rコードのチェックが必要である.

② 80文字読み込むために文字カウンタが必要である.

チェック順であるが，いきなりC/Rコードが入力される場合もあり得るので(いきなり80文字は入力できない)，最初にこれをチェックする．その後，読み込み文字数をチェックする.

C/Rコードをバッファに格納するか否かにより，読み込んだ文字を格納してから終了チェックをするのか，C/Rコードをチェックしてから行動するのかを選択する.

本例ではC/Rコードも1文字と考え，読み込んだ1文字を格納してから2つの終了条件をテストしている.

■各行の意味

行11〜13：D2：C/Rコードも含めた1行の文字カウンタに使用するので，ここでクリアする.

D3：1行の文字数をセット.

A2：読み込んだ文字を格納するバッファ・アドレスの先頭をセット

行15〜19：1文字キーボードからD0に読み込み，D2をカウント・アップし，バッファへ格納する.

行21〜22：2つの終了条件をチェックしている.

22行の意味であるが，次のようなシーケンスが行われている.

① 21行のテスト結果がC/Rコードに等しければEQであるから，DBEQによってBREAK（次の行)へ制御を移行する.

② 21行のテスト結果がC/Rコード以外なら，D3をデクリメントし，
D3＝$FFFF（-1）ならBREAKへ制御を移行
D3≠$FFFF（-1）ならGET_CHARへ分岐

アプリケーション・ヒント

DBcc命令は有用であるので，何か疑問を感じたら，簡単なテスト・プログラムで確認しておくとよい．

図2.32　WHILE_2の様子

・入力文字数−1　→D3
・文字格納アドレス→A3

GET_CHAR　キーボードからD0へ1文字読み込む

D0の内容をBUFFERへ格納

note

D0＝c/r？　YES(ループ脱出)

NO

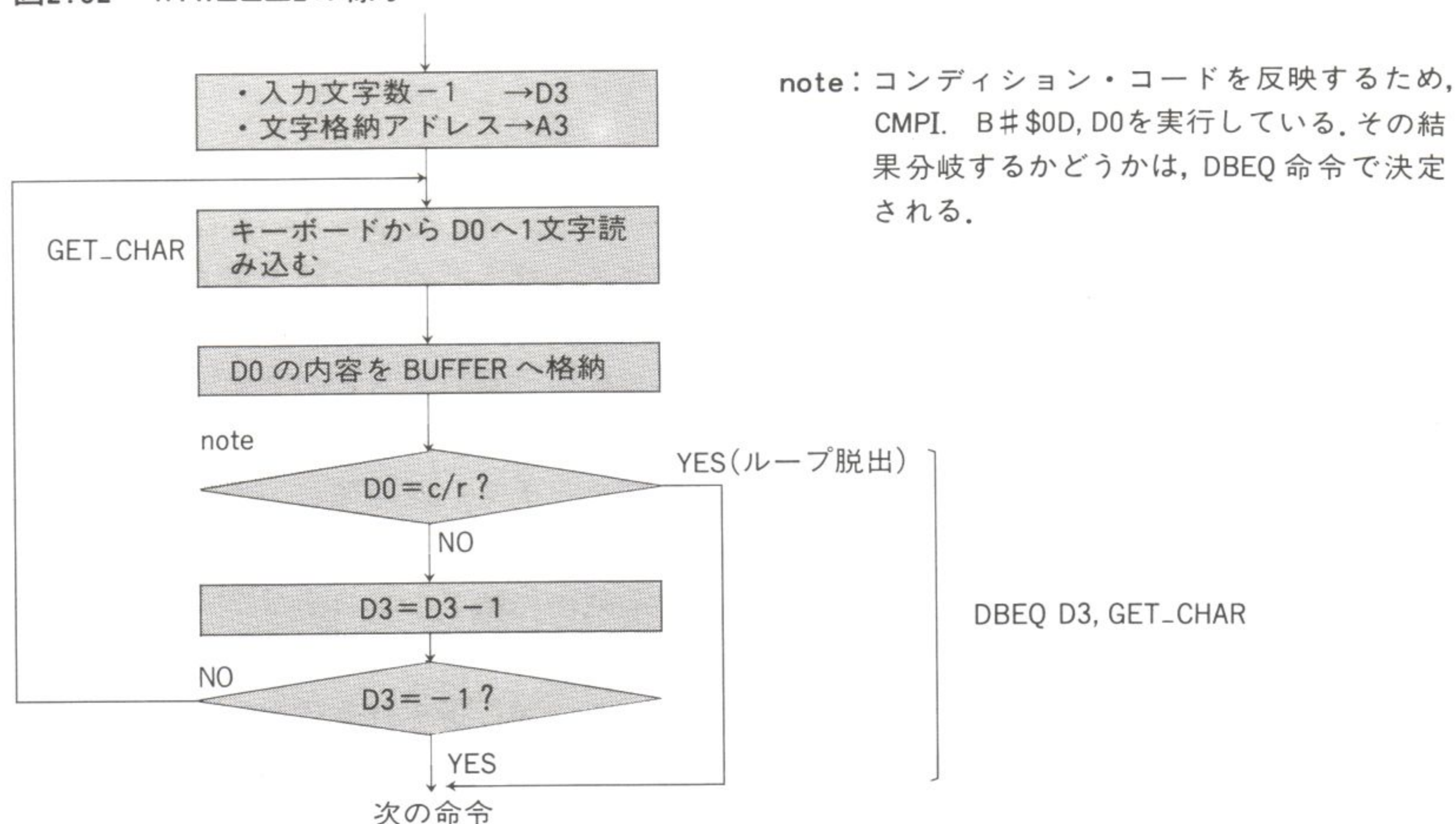
D3＝D3−1

NO　D3＝−1？

YES

次の命令

DBEQ D3, GET_CHAR

note：コンディション・コードを反映するため，CMPI.　B＃$0D, D0を実行している．その結果分岐するかどうかは，DBEQ命令で決定される．

リスト[WHILE_2]

```
LINE ADDR.  CODE/VALUE          LABEL     OP       OPERAND          COMMENT

0005
0006          =00000050         STR_LEN   EQU      80
0007
0009          =00005000                   ORG      $5000
0010          =00005000         WHILE_2
0011 005000 7400                          MOVEQ.L  #0,D2            ;clear len. counter
0012 005002 363C 004F                     MOVE.W   #STR_LEN-1,D3    ;set loop counter to D3
0013 005006 45F9 0000 5020                LEA      BUFFER,A2        ;set buffer address to A3
0014 00500C                     GET_CHAR
0015 00500C 7001                          MOVEQ.L  #1,D0            ;get character
0016 00500E 4E40                          TRAP     #0
0017
0018 005010 5282                          ADDQ.L   #1,D2            ;countup
0019 005012 14C0                          MOVE.B   D0,(A2)+         ;transfer char to buffer
0020
0021 005014 0C00 000D                     CMPI.B   #$0D,D0          ;check c/r code
0022 005018 57CB FFF2                     DBEQ     D3,GET_CHAR
0023 00501C                     BREAK
0025 00501C 7000                          MOVEQ    #0,D0
0026 00501E 4E40                          TRAP     #0
0027
0028                            *                ------------
0029                            *                buffer area
0030                            *                ------------
0031
0032 005020 =00000100           BUFFER    DS.B     256
0033
0034          =00005000                   END      WHILE_2
```

13 サブルーチンとその構成

サブルーチンとは何か，サブルーチン化するときに有用なLINK/UNLK命令の使い方，などについて解説します．

■サブルーチンについて

サブルーチンという概念は，ある程度プログラミングに従事していれば誰でも思いつくことで，要するに，プログラミング作業を効率よく行うための手段と言えます．

サブルーチンとは「ある処理を行うための命令群」で，文字列を表示する命令群をPRINTとすることによって，複数の命令群を「ある名称」で参照するものです．

このようなことを実現するためには，

① 要求される処理を単独で行う部分（サブルーチン）に分類し，その処理を呼び出すために必要な入／出力条件を決定する．

例：文字列出力を行うのであれば，文字列の先頭アドレスをサブルーチンへ渡せばよい．これが入力条件であるが，処理内容によっては結果を知りたいこともあるので，どのような出力がサブルーチンから得られるのか，サブルーチンへの入力／出力情報について検討する．

② 実際にサブルーチンを作成しテストするが，この段階では「こうなるはずだ」という明確な約束に従った処理を行い，様々なケースを想定したテストを十分に行う．

例：＄00で終端する文字列を出力するのであれば，いきなり＄00を取り込んだ場合にも正常であることを確認するとか，これが不便であれば，終端文字を指定する「約束」にサブルーチンの入力条件を変更する．

正しいことを確認したサブルーチンは，それを呼び出すための条件とサブルーチンから返される情報とをメモしておけば，その中での処理ステップを１つ１つ追っていく必要もなく，すっかり忘れてしまってもよい．

以上から，サブルーチン化によるメリットとしては，

① 同じような処理が要求される部分をシンボルで参照できるので，そのつど長々と記述することは不要であるし，何をするプログラムかも一目瞭然である．

② 万一動作しなかった場合でも，プログラム構造が整理されているのでエラー箇所を発見しやすく，１つのミスがプログラム全体に波及する危険性が大変少ない．つまり，１箇所を書き換えることによって，プログラムの最初から最後まで再検討する労力は不要となる．

③ サブルーチンという財産の蓄積により，プログラムすればするほど記述する部分は少なくなる．

④ プログラムの分業化により，複数のプログラマが１つの目的のために作業できる．

これらをひと言で表現すれば，『信頼性の高いプログラムを短期間で完成でき，無駄がない』ということです．

図2.33 サブルーチンの呼び出しとメインへの復帰

〔JSR〜RTS〕

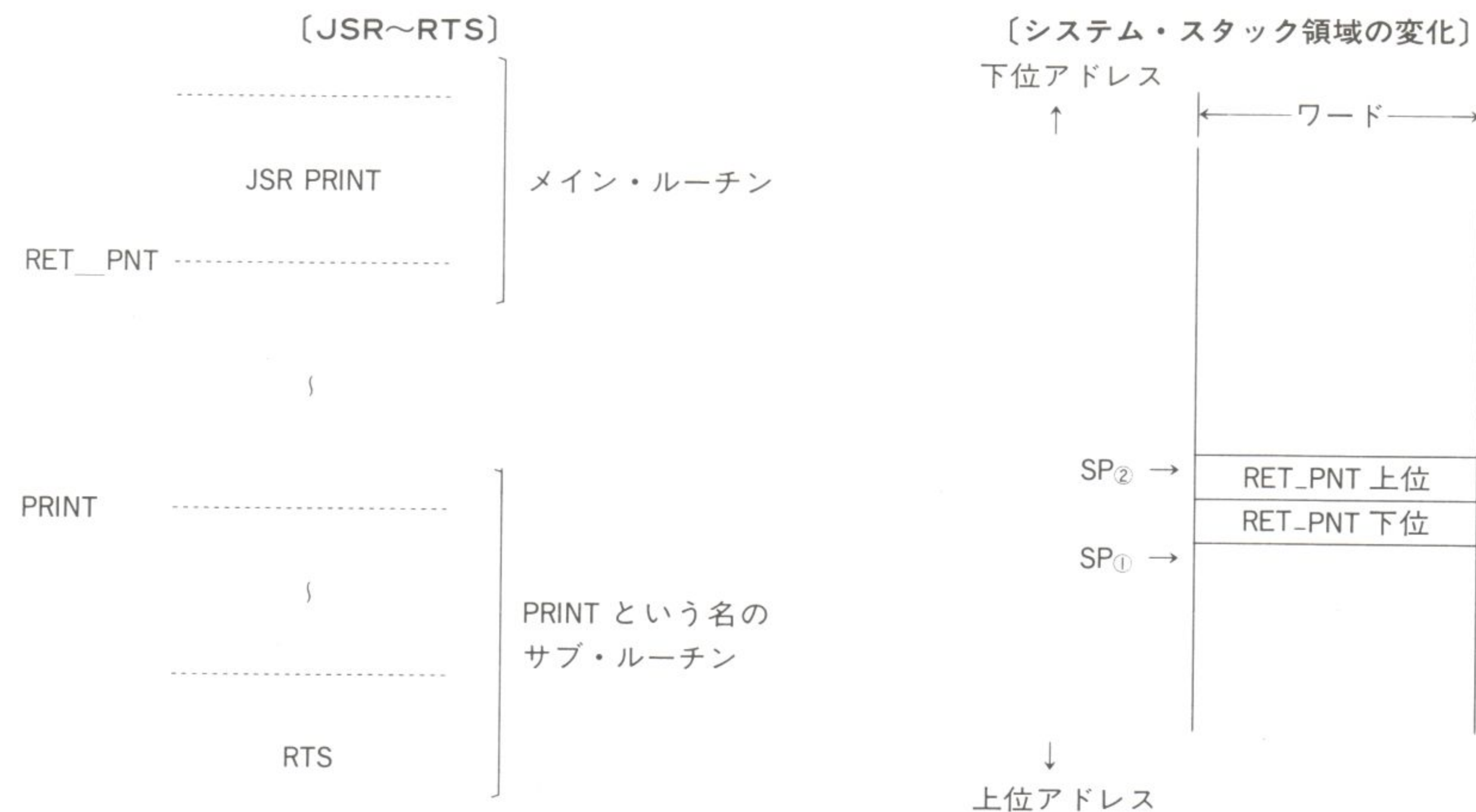
	....	メイン・ルーチン
	JSR PRINT	
RET__PNT	....	
	⁝	
PRINT	....	PRINTという名のサブ・ルーチン
	⁝	
	....	
	RTS	

〔システム・スタック領域の変化〕

下位アドレス ↑

←ワード→

SP② →	RET_PNT 上位
	RET_PNT 下位
SP① →	

↓ 上位アドレス

note：………部は命令が記述されていることを示す.

〔もどりアドレスの管理〕

JSR PRINTによって，PRINTというサブ・ルーチンへ制御が移行するが，スタックは次のように変化する.

① SP①はJSR PRINTという命令を実行する直前の状態であり，JSR PRINTを実行すると，もどり番地であるRET_PNTをシステム・スタックへPUSHする.
それからPRINTというアドレスへ制御を移行する.

② サブ・ルーチンPRINTへエントリした時点のSPはSP②をポイントしており，処理の最後でRTS命令を実行すると，現在のSPのポイントする位置からRET_PNTというもどりアドレスをPOPし，メイン・ルーチンへもどる.

③ メイン・ルーチンでは，何事もなかったかのようにRET_PNTというアドレスから処理を再開する.
このとき，RTSの実行によりシステム・スタックはJSR PRINTを実行する直前の状態に復帰している．すなわちSP①の位置にSPがある.

■サブルーチンの構成

プログラムは長々と記述するのではなく，要求される処理を細部に展開し，それらを1つのサブルーチンに整理することが大切であることを述べましたが，まず『サブルーチンの独立性』ということを強く意識する必要があります.

■サブルーチンの呼び出しとメインへの復帰(BSR/JSRとRTS)

サブルーチン・コール命令BSRやJSRは、BSRやJSR命令の直後に位置している命令が格納されているアドレス（リターン・アドレス）をシステム・スタックへ退避し，サブルーチンへ制御を移行します．サブルーチン側では，RTSやRTRで本来の位置へ制御を移行できるので，プログラマがリターン・アドレスを管理する必要はありません(**図2.33**).

■サブルーチンへのパラメータの渡し方

パラメータの渡し方は重要ですから，少々詳しくこの様子をみてゆくことにします.

サブルーチンへ渡すべきパラメータの数も，サブルーチンに要求される処理によって様々であり，まったく不要であったり，FDD（フロッピー・ディスク・ドライブ）の制御のよ

うに多くのパラメータが必要であることもあります．またOSのドライバ部などはさらに多くのパラメータを渡すだけでなく，復帰情報もそれだけ多くなります．

【方法1：プログラムの一部を用いる】

このような方法でモニタのファンクション・コールを行うシステムもあるようですが，68000という進化したプロセッサでは，このような方法を選択する理由もメリットもありません．

たとえば次のようなプログラムを見つけることがあるかも知れません．

```
JSR   FUNC_1
DC.L  LABEL
MOVE  ……
```

これはFUNC_1という名称のサブルーチンへLABELというシンボルの持つ値を渡している．サブルーチンへエントリしたときには，DC.Lの置かれたアドレスをスタックへ退避するので，そこを参照すれば引数を操作できる．またリターン・アドレスはサブルーチンでRTS命令を実行する直前で，手作業でMOVE……が置かれたアドレスに調整する必要がある．

【方法2：特定の作業用領域を用いる】

プログラム上に一括した作業領域を確保し，この領域をサブルーチンとの通信に使うものです．場合によっては使うこともあるでしょう．

【方法3：データ・レジスタやアドレス・レジスタを用いる】

68000内部にはデータ・レジスタが8本，アドレス・レジスタはA0～A6までの7本（A7はシステム・スタック・ポインタ）が使用可能ですから，これらの豊富なレジスタにパラメータをセットしてサブルーチンを呼び出すことができます．

以下長所，短所に分けてみます．

長所：操作が簡単でプログラム・ステップが少なく高速である．

短所：サブルーチンの独立性が失われる．

サブルーチンの呼び出しに特定のレジスタを使わなければならないので，これがメイン内では大きな制約となることから，きわめて特殊な用途に限定されたサブルーチンに適用すべきで，それ以外にはメリットがない．

たとえば画像処理という1本のプログラム内では何度も呼び出されるが，以後作成するであろうプログラム内で活用するには，何らかの変更をしなければ使用できない．つまり，サブルーチンで使うレジスタのことは常に意識しなければならない．

【方法4：パラメータ・スタックを用いる】

68000には強力なスタック操作が支援されていて，これらを存分に活用しなければなりません．というのは，68000並のスタック操作を従来のマイクロプロセッサ上で実現するには，いくつかの面倒な手続きが必要だからです．換言すれば，68000ユーザは大変に恵まれた環境にあるわけです．

スタックを操作するには以下のアドレッシングを使用します．

-(SP)：スタックへのプッシュを行うので，下記の例ではD0の下位ワードをスタック領域へ転送する．

```
MOVE.W D0,-(SP)
```

(SP)+：スタックからのポップを行うので，下記の例ではスタック領域からD0の下位ワ

図2.34　パラメータ・スタックの様子

〔コール側〕

```
        MOVE.L   D0,-(SP)
        JSR      FUNC
RET__PNT
        次の命令
```

〔サブルーチンへエントリしたときのスタック〕

下位アドレス ↑ ←ワード→

システム・スタック・エリア

SP③ →	+0	RET-PNTの上位ワード	メインへのリターン・アドレス
	+2	RET-PNTの下位ワード	
SP② →	+4	D0の上位ワード	メイン（コール側）から渡された引数
		D0の下位ワード	
SP① →			

サブルーチン側では
MOVE.L　4(SP),Dn
のようにして任意のデータ・レジスタへ引数をロードすることができる．

↓ 上位アドレス

SP①：MOVE.L D0,－(SP) を実行する直前のSP
SP②：MOVE.L D0,－(SP) 実行後のSP
SP③：JSR FUNCを実行し，FUNCというサブルーチンへ分岐して来たときのSP

ードへの転送である．

```
MOVE.W (SP)+,D0
```

※これらの意味がよく理解できない場合は，アドレッシング・モードの説明を参照してください．

●具体的なパラメータ授受の様子

FUNCという名のサブルーチンが何をするかは別として，とにかくFUNCはデータを加工してその結果を返してくるものとしましょう．FUNCへ渡すべきデータがD0のロング・ワードに格納されていれば，以下のようにしてD0の内容をサブルーチンへ渡します．

```
         MOVE.L D0,-(SP)     :D0を引数としてスタックへプッシュ
         JSR    FUNC         :サブルーチンFUNCへ制御を移行
RET__PNT
         次の命令 ……
```

サブルーチン側では引数を受け取り，その結果を同じロケーションへ返さなければなりません．サブルーチンへエントリした時点のSPの変化は**図2.34**に示す通りです．

したがって次のようにすれば引数をD0へ取り込むことができます．

```
MOVE.L 4(SP),D0    :別にD0でなくてもよい
```

しかしサブルーチン側にとっては，以下のような問題が未解決のまま残ります．

① このままではレジスタをスタックへ退避したり，このルーチンから別のサブルーチンを呼び出すことが不便である．
つまりサブルーチンFUNCが使用するスタック領域が適切ではなく，パラメータの存在する領域を破壊しないように，一時的にSPを書き換える必要がある．

② 場合によっては作業用のメモリ領域（ローカル・エリア）も必要である．

一般にはサブルーチンへエントリした時点のSPを作業用レジスタへコピーし，フリーになったSPを下位アドレス方向へ移動する作業を行い，サブルーチンからもどるときにSPを元の位置へ復元します．

68000ではこのような操作を効果的に行うことができるLINKとUNLKという命令があります．

●LINK/UNLK命令とパラメータの授受

LINK命令はシステム・スタック上に32767バイトまでのローカル・エリアを確保し，UNLKは確保したローカル・エリアを開放するものですが，「ローカル・エリアとは何か」ということよりも，まず実例を示すことにします．

ただし以下の事柄に注目すべきです．

① LINK/UNLKはサブルーチン側で意味を持つ命令である．

② システム・スタック領域はSPによって管理され，DC命令やDS命令で確保した領域と異なり，どのようなサブルーチンからも共通にアクセスできる唯一の領域である．

先のように呼び出されるサブルーチンFUNC内では，一般的な手続きとして以下のように記述しますが，これらの命令とスタックの変化との関係は**図2.35**に示す通りです．

図2.35
LINK/UNLKとスタックの変化

ワード

SP② → SR
SP① → RET_PNT（もどり番地）
D0(引数)

LINK 実行前

SP① ：サブルーチン FUNC へ分岐して来たときの SP.

SP② ：MOVE. W SR, −(SP)により SR をスタックへプッシュしたときの SP.
これは LINK を実行する直前の SP となる.

ワード

システム・スタック・エリア

SP → −16
−14
−12
−10
−8
−6
−4
−2
LINK A6, ♯−16によって確保された16バイトのローカル・エリア
A6 → +0
+2
旧 A6
+4 SR
+6
+8
RET_PNT（もどり番地）
+10
+12
D0(引数)

LINK 実行後

LINK A6, ♯−16により，図のようにA6, SP が決定される．つまり，

1: まず LINK 命令で指定されている A6 をスタックへ退避する．
（このときSPは−4されるので、図のA6に位置している．）

2: A6 は退避されたので SP の内容を A6 に転送する．これが図の A6 である．

3: LINK 命令で指定された −16 を SP に加える．これが図の SP である．

ワード

SP → MOVEM によって退避されたレジスタ群
−16
−14
−12
−10
−8
−6
−4
−2
LINK A6, ♯−16によって確保された16バイトのローカル・エリア
A6 → +0
+2
旧 A6
+4 SR
+6
+8
RET_PNT（もどり番地）
+10
D0(引数)

MOVEM 実行後

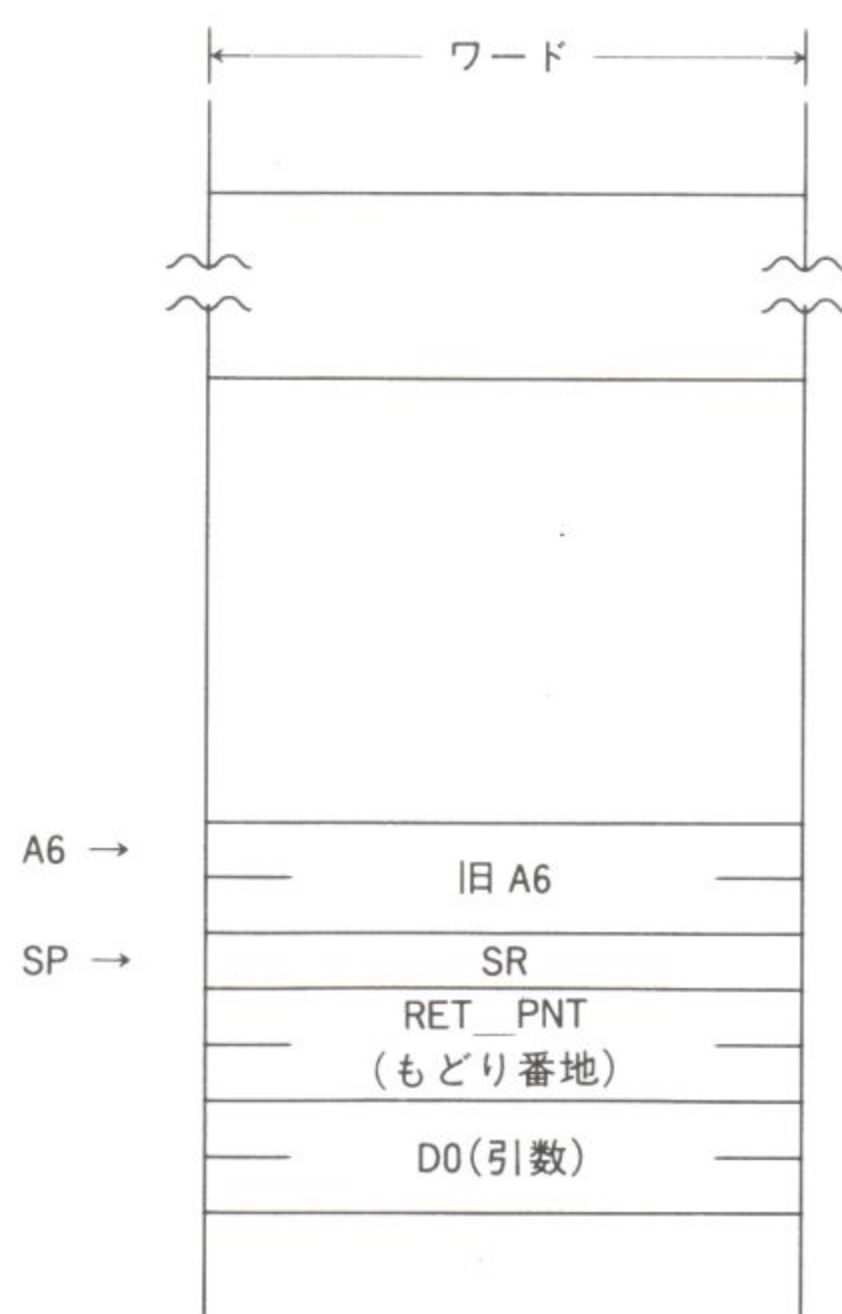
SP：MOVEM によりスタックへレジスタ群がプッシュされるので，それに応じて SP は更新される．

```
FUNC
        MOVE.W      SR,-(SP)                 ：SRをスタックへ退避
        LINK        A6,#-16                  ：ローカル・エリアを16バイト確保
        MOVEM.L     D0-D7/A0-A5,-(SP)        ：レジスタ群をスタックへ退避
        ─────────────────────────
        必要な処理を行うが，ここでは
            MOVE.L   10(A6),D4               ：別にD4でなくてもよい
        のように引数をアクセスできる
        ─────────────────────────
        MOVEM.L     (SP)+,D0-D7/A0-A5        ：レジスタ群をスタックから復帰
        UNLK        A6                       ：ローカル・エリアの開放
        RTR                                  ：SRをスタックから取り出し，メインへ復帰
```

さてRTRで無事メイン・ルーチンへ復帰した時点では，スタックはD0をプッシュした状態にあり，次の2通りのいずれかの命令を実行してSPを復元しなければなりません．

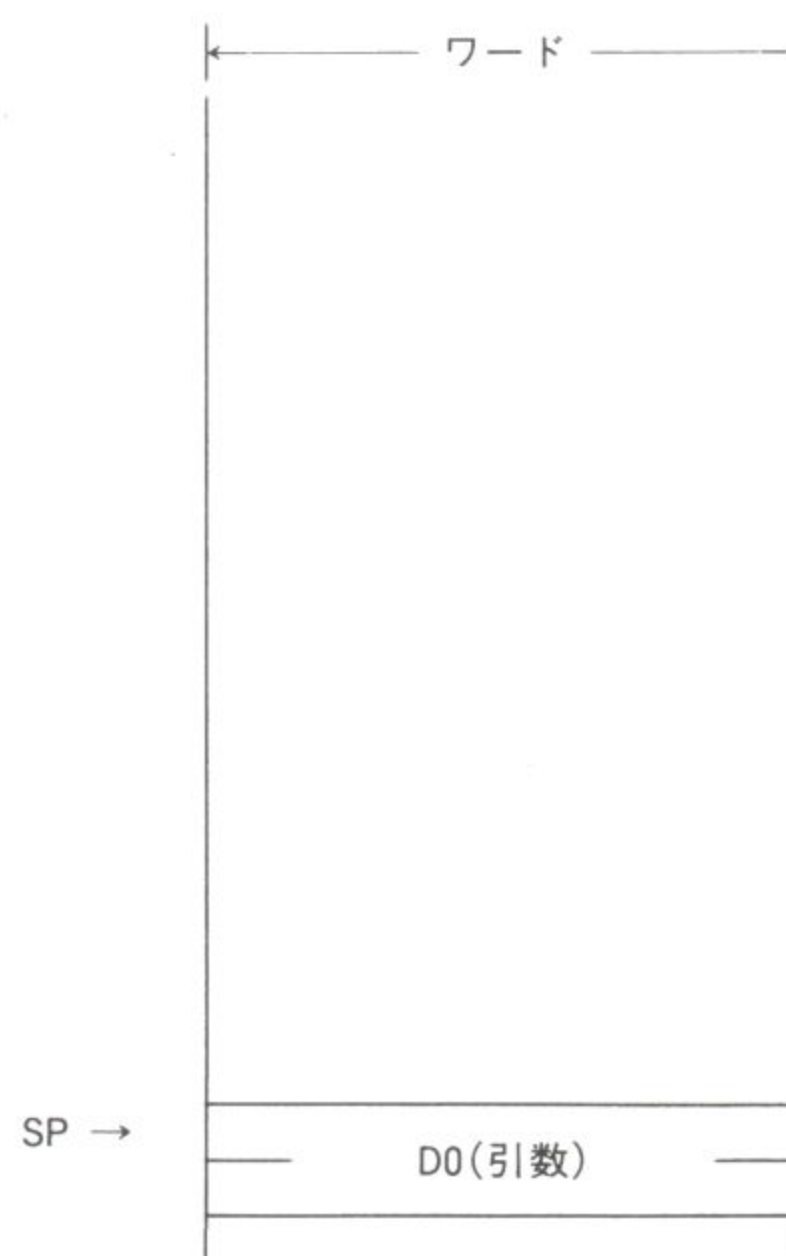
```
●  MOVE.L    (SP)+,D2    ：FUNCからの結果を例えばD2へポップ
     ↕          ↕
●  ADDQ.L    #4,SP       ：スタックを強制的に調整（復元）
  (ADDA.L    #4,SP)
```

ワード

A6 →
旧A6
SP →
SR
RET__PNT
(もどり番地)
D0(引数)

UNLK 実行後

UNLK A6 により
1：A6の内容がSPに転送される．これでA6はフリーになる．
2：SPでポイントされるアドレスからロングワード(旧A6)をUNLK命令で指定したアドレス・レジスタへポップする．
これでA6はサブルーチンへ分岐したときの内容へ復元され，SPは＋4される．これが図のSPである．

ワード

SP →
D0(引数)

RTR 実行後

プログラム制御はRET__PNTへ移行するが，SPはJSRを実行する直前にもどり，スタックには引数がプッシュされたままになっている．

このようにLINK/UNLK命令は大変便利な命令であり，プログラマの必修事項です．

Cコンパイラに精通されていれば，いかに効果的な引数の授受を少ないステップで実行できるかを理解できるはずですし，LINK/UNLKは高級言語指向の命令セットであるわけです．

ここでLINK/UNLK命令のポイントを整理しておきます．

LINK An,#<data> ：Anで指定したアドレス・レジスタで引数の操作を行い，ディスプレースメント付きアドレス・レジスタ間接形式を採用する．
たとえばA6を指定すれば，

disp(A6)

のようになり，dispの値はメインからの引数のサイズや個数に依存する（本例では10）．また指定したAnは重要なアドレス・ポインタでもあるので，安易に破壊しないように注意する．
#<data>でサブルーチン側で使用するローカル・エリアを32767まで指定できるが，必ず負（マイナス）の値を指定しなければ無意味である．この大きさは処理内容に依存する．

UNLK An ：LINKで指定したAnを指定する必要がある．

■渡される内容のもつ意味

スタックへプッシュされサブルーチンへ渡される内容は，単なるデータとポインタに大別でき，サブルーチンからの復帰情報も同様です．これらは処理内容を解析すれば容易に選択できるものであり，どのような処理をサブルーチンに望むか，ということが先決問題です．

たとえば，文字列のソート（整列）を実行するのに，スタック領域へ個々の文字列を転送してからサブルーチンを呼び出すよりは，文字列が格納されているアドレス(ポインタ)を渡した方が便利に決まっています（スタック空間は限られた記憶空間なので，すべての文字列を転送できるかどうかも疑問である）．

APPENDIX◉ローカル・エリア

サブルーチンという概念がなければローカル・エリアという記憶領域も存在しないが，サブルーチンという考え方は大変重要であり，しかもサブルーチンは完全なブラック・ボックスとして機能する必要がある．つまり，メイン～サブルーチン間での引数の授受に特定の記憶領域を割り当てたり，作業用にメインで使用する記憶空間を割り当てるべきではない．

サブルーチンが使用する記憶領域は必要に応じてサブルーチン側で確保すべきで，サブルーチンからリターンするときに開放すればよい．このように必要なときだけ一時的に確保される記憶領域をローカル・エリアと呼び，この空間はシステム・スタック上に確保される．

1 メイン～サブルーチン間での引数の授受

◉サンプルプログラム ［ARG］

3つの引数（バイト，ワード，ロング・ワード）をスタックを介してサブルーチンへ渡し，サブルーチン側で正しくこれらの内容をアクセスできることを確認します．

■解法

サブルーチンの先頭でSRをスタックへ退避する場合，最後にプッシュされた引数はリンク・ポインタの先頭から10進んだ地点で，SRをスタックへ退避しなければ，リンク・ポインタから8進んだ地点となる．

■各行の意味

行9～11：順に，ロング・ワード，ワード，バイトの各データをスタックへプッシュする．

行12　：サブルーチンARG_DMPをコールしている．

行13　：サブルーチンから返される値がない場合に必要な操作で，行9～11で更新されたSPを元の状態に復元している．

行14　：REEAK_0はサブルーチンによってスタックが復元されていることを確認するためのブレーク・ポイントである．

行23～25：サブルーチン側で必要な初期化で，SRの退避，スタック領域のリンク，レジスタ群の退避を行うが，必ずこの順でなければならない．ローカル・エリアを確保する必要はないので，24行のようにLINK命令を使用した．

行27～29：次のように引数が各レジスタへ転送される．

$55　→　D2
$A0A0　→　D1
$FFFFFFFF　→　D0

（この様子は，$5030番地でプログラムをブレークし，その時のD2，D1，D0の内容をチェックすれば確認できる）

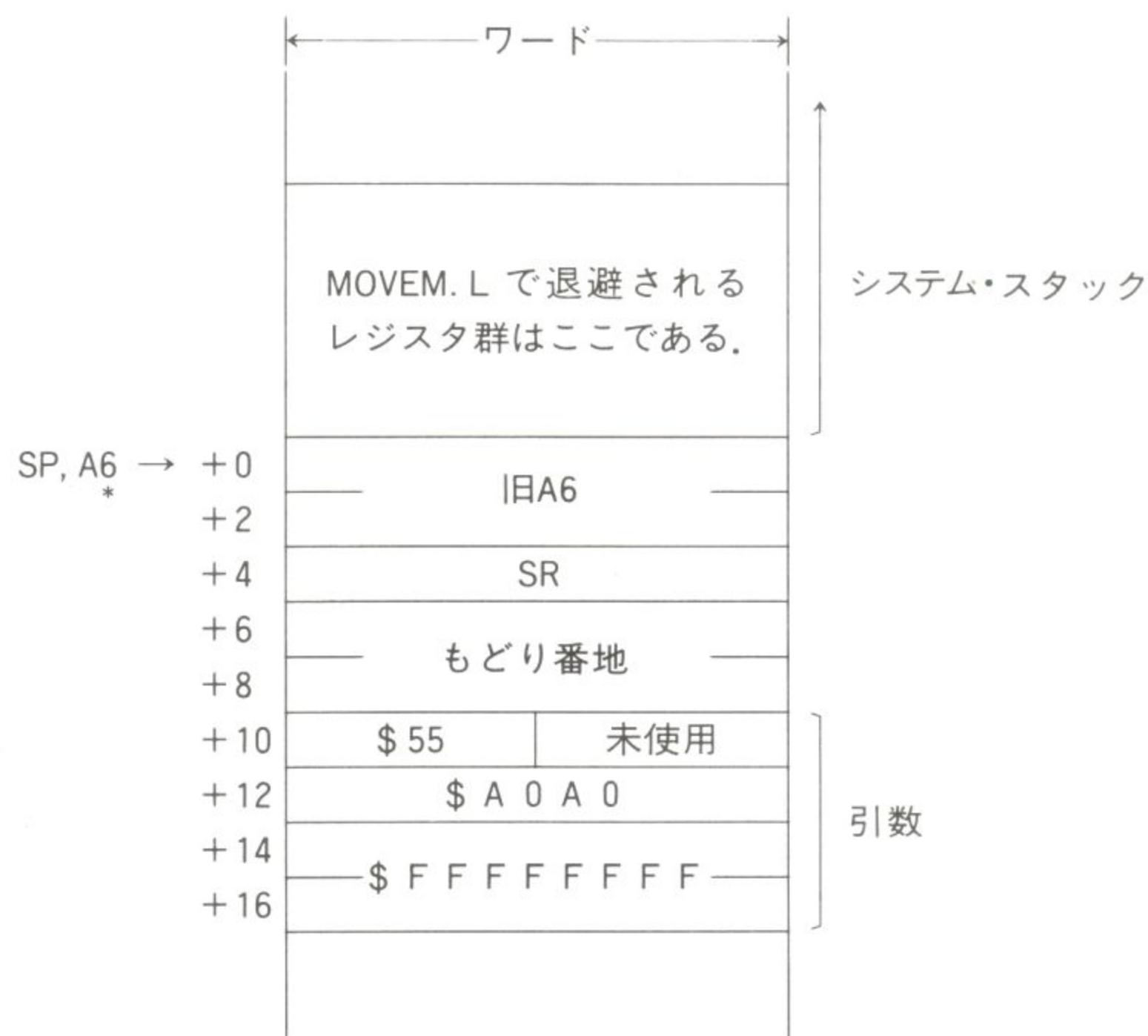
行31～33：リターン時に必要な手続きで行23～25とは逆の操作を行っているが，必ずこの順でなければならない．

以上によるスタックの様子については**図2.36**をご覧いただきたい．

■注意事項

バイト・データがシステム・スタックへプッシュされてもSPは2だけ減じられる．このときにバイト・データが転送されるロケーションは偶数アドレスなので，サブルーチン側で操作する引数も偶数アドレスとなる．27行がそれである．

なおワード・データがスタックへプッシュされ，サブルーチン側でバイト単位で引数をアクセスすることも，もちろん可能である．

図2.36　ARGのスタック図

リスト[ARG]

```
LINE ADDR.  CODE/VALUE          LABEL     OP        OPERAND            COMMENT

0005
0007        =00005000                   ORG       $5000
0008        =00005000          ARG
0009 005000 2F3C FFFF FFFF                MOVE.L    #$FFFFFFFF,-(SP)
0010 005006 3F3C A0A0                     MOVE.W    #$A0A0,-(SP)
0011 00500A 1F3C 0055                     MOVE.B    #$55,-(SP)
0012 00500E 4EB9 0000 501A                JSR       ARG_DMP
0013 005014 508F                          ADDQ.L    #8,SP              ;adjust stack pointer
0014 005016                    BREAK_0
0016 005016 7000                          MOVEQ     #0,D0
0017 005018 4E40                          TRAP      #0
0018
0019                           *                  ---
0020                           *                  sub
0021                           *                  ---
0022 00501A                    ARG_DMP
0023 00501A 40E7                          MOVE.W    SR,-(SP)
0024 00501C 4E56 0000                     LINK      A6,#0
0025 005020 48E7 FFFC                     MOVEM.L   D0-D7/A0-A5,-(SP)
0026
0027 005024 142E 000A                     MOVE.B    10(A6),D2
0028 005028 322E 000C                     MOVE.W    12(A6),D1
0029 00502C 202E 000E                     MOVE.L    14(A6),D0
0030 005030                    BREAK_1
0031 005030 4CDF 3FFF                     MOVEM.L   (SP)+,D0-D7/A0-A5
0032 005034 4E5E                          UNLK      A6
0033 005036 4E77                          RTR
0034
0035        =00005000                     END       ARG
```

2 ローカル・エリアを使用した文字数のカウント

◉サンプルプログラム ［C__CNT］

文字数のカウント程度では特にローカル・エリアは必要とされませんが，スタック上に確保したローカル·エリアを作業用のメモリ·レジスタに割り当てて文字数をカウントしています．

■動作

引数として$00で終了する文字列のアドレス（ポインタ）を渡し，サブルーチンからは文字数を返してくる．データ・サイズは双方ともにロング・ワードである．

■各行の意味

行 9～11：CCNT__SUBへ引数を渡し，結果をD5で受けている．

行20～23：SRの退避，ローカル・エリアを2バイト確保，使用レジスタの退避を行う．

行25～26：引数をA0に格納し，文字数をカウントする作業用ローカル·エリアをクリアする．

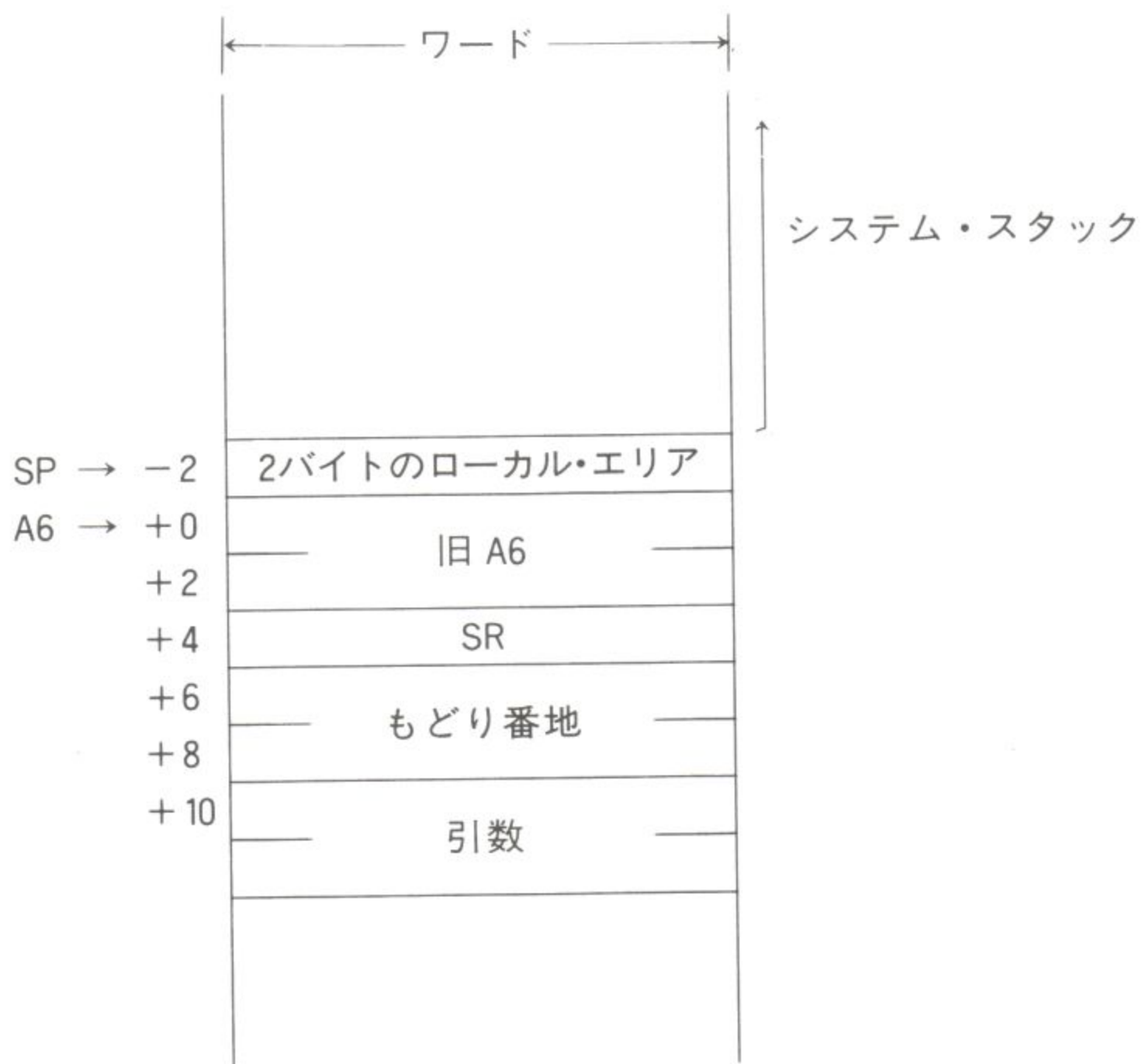
アプリケーション・ヒント

引数，ローカル・エリアへのアクセスはリンク・ポインタに指定したA6を使用しているが，複雑な処理では，LEA命令を使って別々の先頭アドレスに分離してから作業を開始すればよい．

図2.37
C__CNTのスタック図

SP →	−2	2バイトのローカル·エリア
A6 →	+0	旧 A6
	+2	
	+4	SR
	+6	もどり番地
	+8	
	+10	引数

ワード

システム・スタック

●SP, A6の位置は，LINK 命令後の状態である．
●ローカル・エリアおよび引数へのアクセスは A6 で行うことができる．

行27〜31：$00が見つかるまでの文字数をローカル・エリア上のメモリ・カウンタに求める．

行33　　：求めたカウンタ値を引数エリアに返している．本例ではローカル・エリアの実例としたが，カウンタを最初から引数エリアに使用すれば，本行は不要である．

行35〜37：リターン時の手続きをしている．

リスト[C_CNT]

```
LINE ADDR. CODE/VALUE          LABEL     OP       OPERAND          COMMENT

0005
0007          =00005000                  ORG      $5000
0008          =00005000        C_CNT
0009 005000 4879 0000 503C               PEA      DATA
0010 005006 4EB9 0000 5012               JSR      CCNT_SUB
0011 00500C 2A1F                         MOVE.L   (SP)+,D5         ;D5 = character count
0012
0014 00500E 7000                         MOVEQ    #0,D0
0015 005010 4E40                         TRAP     #0
0016
0017                           *                  --------------------
0018                           *                  character count sub.
0019                           *                  --------------------
0020 005012                    CCNT_SUB
0021 005012 40E7                         MOVE.W   SR,-(SP)
0022 005014 4E56 FFFE                    LINK     A6,#-2
0023 005018 48E7 8080                    MOVEM.L  D0/A0,-(SP)
0024
0025 00501C 206E 000A                    MOVE.L   10(A6),A0        ;load pointer to A0
0026 005020 42AE FFFE                    CLR.L    -2(A6)           ;clear counter
0027 005024                    CNT_LOOP
0028 005024 1018                         MOVE.B   (A0)+,D0         ;read character
0029 005026 6706                         BEQ      LD_CNT
0030 005028 52AE FFFE                    ADDQ.L   #1,-2(A6)        ;countup
0031 00502C 60F6                         BRA      CNT_LOOP
0032 00502E                    LD_CNT
0033 00502E 2D6E FFFE 000A               MOVE.L   -2(A6),10(A6)
0034
0035 005034 4CDF 0101                    MOVEM.L  (SP)+,D0/A0
0036 005038 4E5E                         UNLK     A6
0037 00503A 4E77                         RTR
0038
0039                           *                  ---------
0040                           *                  data area
0041                           *                  ---------
0042
0043 00503C 6162 6364 6566     DATA      DC.B     "abcdefghijkl",0
            6768 696A 6B6C
            00
0044 005049
0045          =00005000                  END      C_CNT
```

3 ポインタ(アドレス)を取り出すサブルーチン

◉サンプルプログラム [OBT__PNT]

引数としてポインタを格納してある表の先頭アドレスとインデックス番号をサブルーチンへ渡し，表を参照するサブルーチンを考えてみます．

■解法

① サブルーチンはSUB__0～SUB__3までの4つを用意し，ここへ制御が移行したことを確認するためD1に特定の値をセットするものとする．

② SUB__0～SUB__3の位置するアドレスを格納した表をSUB__TBLから定義する．

③ 後はインデックスとSUB__TBLをサブルーチンGET__APNTへ渡し，必要なアドレスを求め，そこをサブルーチン・コールすれば，SUB__0～SUB__3までのいずれかを呼び出すことができるはずだ．

■各行の意味

行9～11：引数としてD0に格納されたインデックス(0～3),SUB__TBLのアドレスをスタックへプッシュし，GET__APNTを呼び出す．

行13～16：結果の受け取りとテストを行う．

行13　：A5にSUB__0～SUB__3までのいずれかのサブルーチンの先頭アドレスが求められる．

行14　：GET__APNTからはポインタしか返されないので，SPを調整する．

行15　：A5にセットされたアドレスをサブルーチン・コールするので，9行でD0がゼロならA5には$5042が代入され，

```
JSR (A5)
```

によって$5042をサブルーチン・コールする．

行16　：サブルーチンPUT__CHRはD1にセットされた文字コードをスクリーンへ表示するので，これまでの結果を確認できる．

行25～38：2つの引数を受け取り，必要な表を参照するサブルーチンである．

行25～28：サブルーチンの先頭で行うべき処理部である．

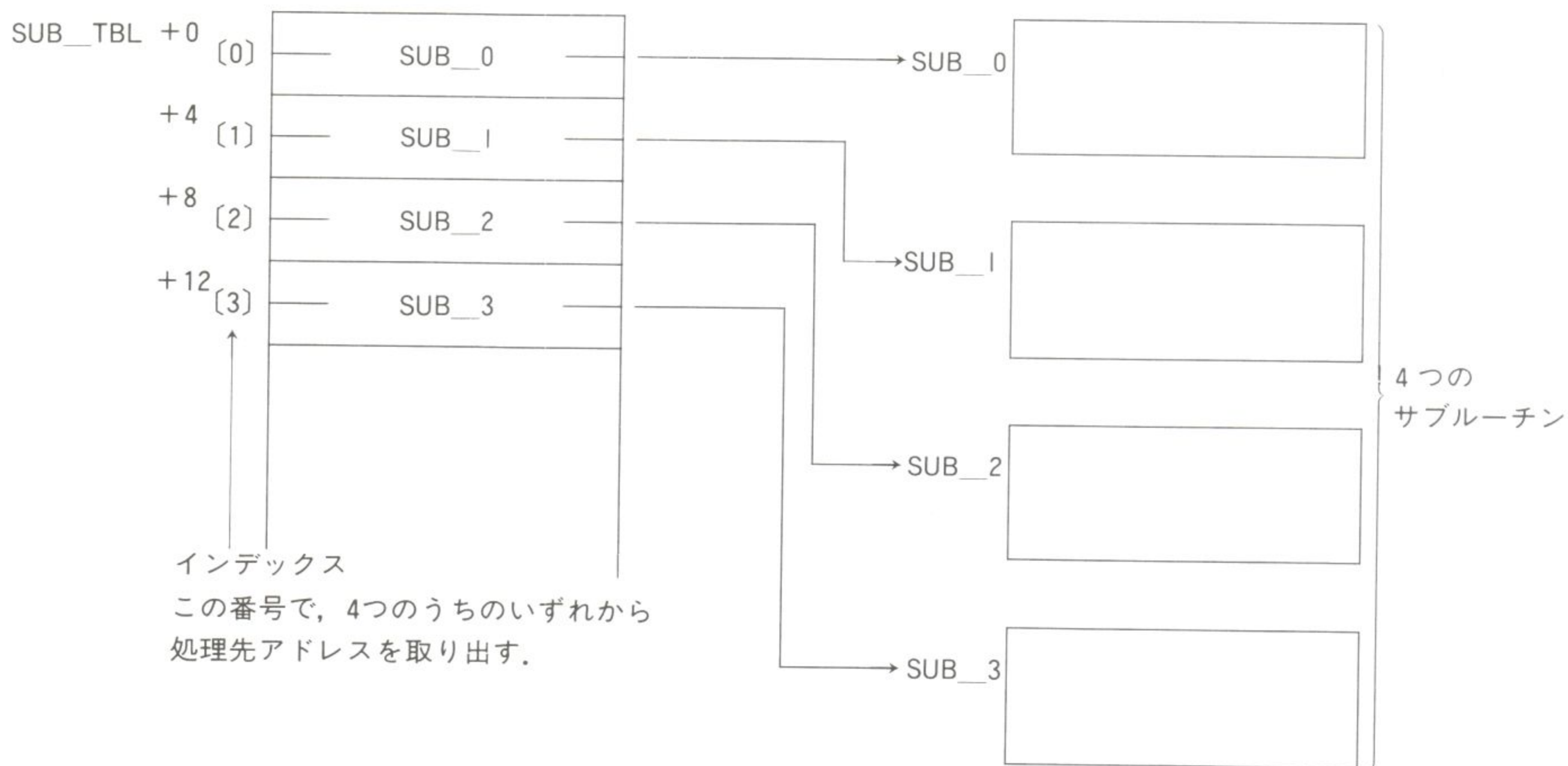
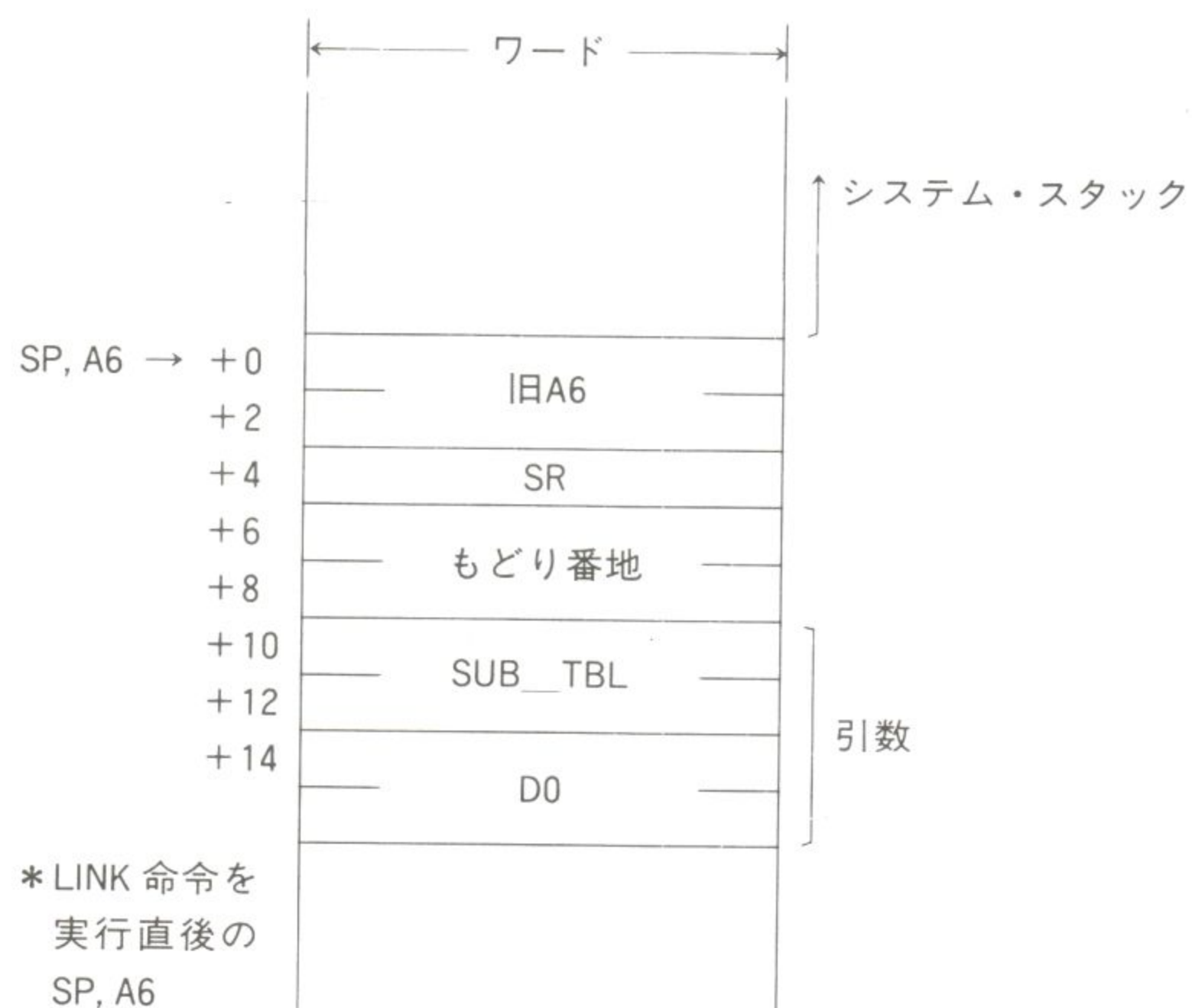
行30～34：引数を受け取り，結果を所定位置へ返す．
33，34は以下のように1行で記述することもできる．

```
MOVE.L  0(A0,D0.L),10(A6)
```

行36～38：リターン時に必要な処理を行う．

図2.38　OBT__PNTのスタック図

ワード
システム・スタック
SP, A6 → +0
+2
旧A6
+4
SR
+6
+8
もどり番地
+10
+12
SUB__TBL
+14
D0
引数
＊LINK命令を
実行直後の
SP, A6

SUB__TBL +0 〔0〕 SUB__0 → SUB__0
+4 〔1〕 SUB__1 → SUB__1
+8 〔2〕 SUB__2 → SUB__2
+12 〔3〕 SUB__3 → SUB__3
4つの
サブルーチン
インデックス
この番号で，4つのうちのいずれから
処理先アドレスを取り出す．

リスト[OBT__PNT]

```
LINE ADDR. CODE/VALUE          LABEL    OP       OPERAND      COMMENT

0005
0007        =00005000                   ORG      $5000
0008        =00005000          OBT_PNT
0009 005000 2F00                        MOVE.L   D0,-(SP)     ;push index
0010 005002 4879 0000 5060              PEA      SUB_TBL      ;push base address
0011 005008 4EB9 0000 501E              JSR      GET_APNT
```

```
0012 00500E                  RET_PNT
0013 00500E 2A5F                     MOVE.L   (SP)+,A5        ;get address
0014 005010 588F                     ADDQ.L   #4,SP           ;adjust stack pointer
0015 005012 4E95                     JSR      (A5)            ;setup D1
0016 005014 4EB9 0000 505A           JSR      PUT_CHR
0017
0019 00501A 7000                     MOVEQ    #0,D0
0020 00501C 4E40                     TRAP     #0
0021 00501E
0022                         *                -----------
0023                         *                get pointer
0024                         *                -----------
0025 00501E                  GET_APNT
0026 00501E 40E7                     MOVE.W   SR,-(SP)        ;push SR
0027 005020 4E56 0000                LINK     A6,#0
0028 005024 48E7 8080                MOVEM.L  D0/A0,-(SP)     ;push register list
0029
0030 005028 206E 000A                MOVE.L   10(A6),A0       ;load base address to A0
0031 00502C 202E 000E                MOVE.L   14(A6),D0       ;load index to D0
0032 005030 E588                     LSL.L    #2,D0           ;D0=D0*4
0033 005032 2070 0800                MOVE.L   0(A0,D0.L),A0   ;pickup pointer to A0
0034 005036 2D48 000A                MOVE.L   A0,10(A6)       ;load A0 to stack area
0035
0036 00503A 4CDF 0101                MOVEM.L  (SP)+,D0/A0     ;pop register list
0037 00503E 4E5E                     UNLK     A6
0038 005040 4E77                     RTR
0039
0040                         *                --------
0041                         *                for test
0042                         *                --------
0043
0044 005042 123C 0030        SUB_0   MOVE.B   #'0',D1
0045 005046 4E75                     RTS
0046
0047 005048 123C 0031        SUB_1   MOVE.B   #'1',D1
0048 00504C 4E75                     RTS
0049
0050 00504E 123C 0032        SUB_2   MOVE.B   #'2',D1
0051 005052 4E75                     RTS
0052
0053 005054 123C 0033        SUB_3   MOVE.B   #'3',D1
0054 005058 4E75                     RTS
0055
0056                         *                -----------------------
0057                         *                put character to screen
0058                         *                -----------------------
0059 00505A                  PUT_CHR
0060 00505A 7002                     MOVEQ.L  #2,D0           ;function 2
0061 00505C 4E40                     TRAP     #0              ;printout D1.B to screen
0062 00505E 4E75                     RTS
0063
0064                         *                -------------
0065                         *                address table
0066                         *                -------------
0067
0068 005060 0000 5042        SUB_TBL DC.L     SUB_0
0069 005064 0000 5048                DC.L     SUB_1
0070 005068 0000 504E                DC.L     SUB_2
0071 00506C 0000 5054                DC.L     SUB_3
0072
0073        =00005000                END      OBT_PNT
```

14 配列

ここでは２次元配列の扱い方について説明しますが，３次元以上の配列も同様な考えかたで対処できます．

■２次元配列の考え方

ARRAY（a，b）のような２次元配列にはインデックスが２つあり，１番目の添え字を行（Column），次の添え字を列（Row）といい，ｂ個のデータの集まりを行と表現し，全体では，ｂ個で構成される行がａ行あることになります．

配列をアクセスするにはその配列のアドレスを計算すればよいので，そのためには以下のように考えるわけです．

ARRAY(a,b)　：a＝0，1，…　a－1　b＝0，1，…　b－1

ARRAY（a，b）の（　）内の添え字a/bと，アクセスされる場所の移動量との関係は，

a（行）はアクセス場所を大きく移動する役目
b（列）はアクセス場所を細かく移動する役目

なので，まずａから行の先頭アドレスを算出し，次に各行の先頭からの隔たりをｂから求めます．これらを加算すれば，配列ARRAYがゼロ番地から定義された場合のアドレスが求められます．

① まずｂ個のデータが集まって１行を構成するので，行サイズが求められる．

行サイズ＝（列の構成個数）×（データ・サイズ）

② 各行の先頭オフセットは行サイズ分増加する．

行の先頭オフセット＝（行番号）×（行サイズ）

③ 各行からの隔たり（列オフセット）はデータ・サイズと列番号から求められる．

列のオフセット＝（列番号）×（データ・サイズ）

④ 実際のアドレス（求めるアドレス）は次のようになる．
配列のアドレス＝（配列のベース・アドレス）＋（行の先頭オフセット）
＋（列のオフセット）

■３次元配列の考え方

３次元配列では，２次元配列が複数個集まったものであると考えます．
そこで

ARRAY（a，b，c）

なら，アドレスを大きく変化させる部分はａであるので，（b，c）がａページ，あるいはａセット集まったものであると考えればよく，多次元配列もこのようにして容易に拡張することができます．

1 2次元配列のアクセス

◉サンプルプログラム ［ARRAY］

ここではバイト，ワード，ロング・ワードの各サイズを格納する2次元配列をアクセスするための汎用サブルーチンを作成します．

■動作

〝解法〟の項で説明されている5つのパラメータをサブルーチンOBT_ARYに渡すと，必要な配列のアドレスの計算結果を返して来るが，パラメータを変更することにより，どのようなデータ・サイズにも対応できるようにした．

なお本配列では，7行6列，つまり高級言語ではARRAY（7，6）のように宣言されているものとしたが，100列のロング・ワードから構成されれば，行サイズを400（100＊4）とすればよいだけのことである．

■解法

配列のアドレスを決定するためには，以下のようなパラメータを必要とする．

① 配列の先頭（ベース）アドレス
② 配列の1行が何バイトで構成されるかという値（行サイズ）
③ 行番号
④ 列番号
⑤ データ・サイズの識別子（0：バイト，1：ワード，2：ロング・ワード）

■各行の意味

行26〜30：パラメータをスタックへプッシュしている．これらの命令を実行するためには，すでに適切な値が所定のレジスタへ格納されていなければならない．

行31：OBT__ARYをコール．

行33〜34：スタックを調整し，結果を受け取っている．

行35：ここでプログラムをブレークし，所定のアドレスが計算されたか否かを確認している．

行43〜65：サブルーチンOBT__ARYである．渡されたパラメータから配列のアドレスを計

アプリケーション・ヒント

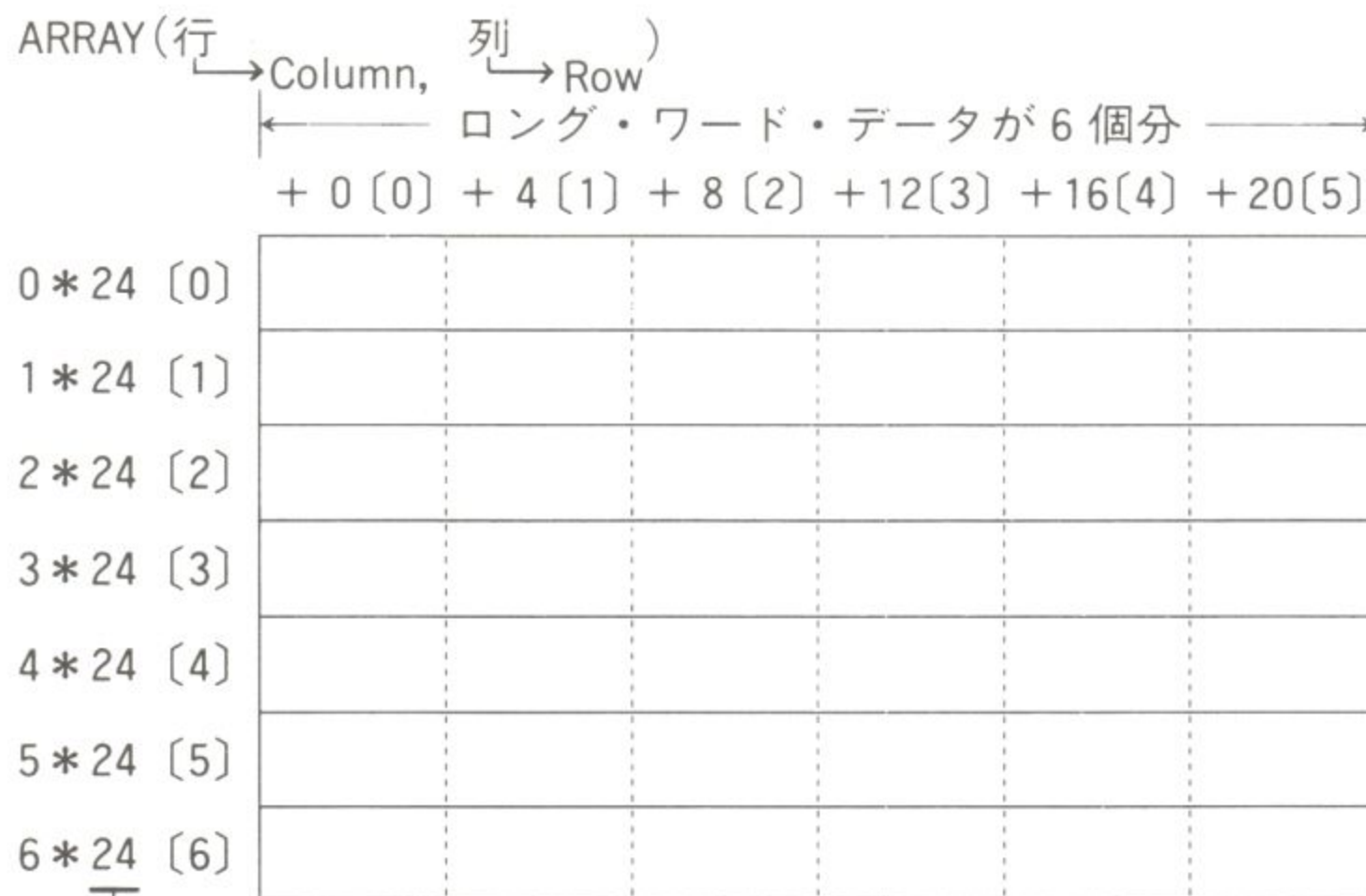
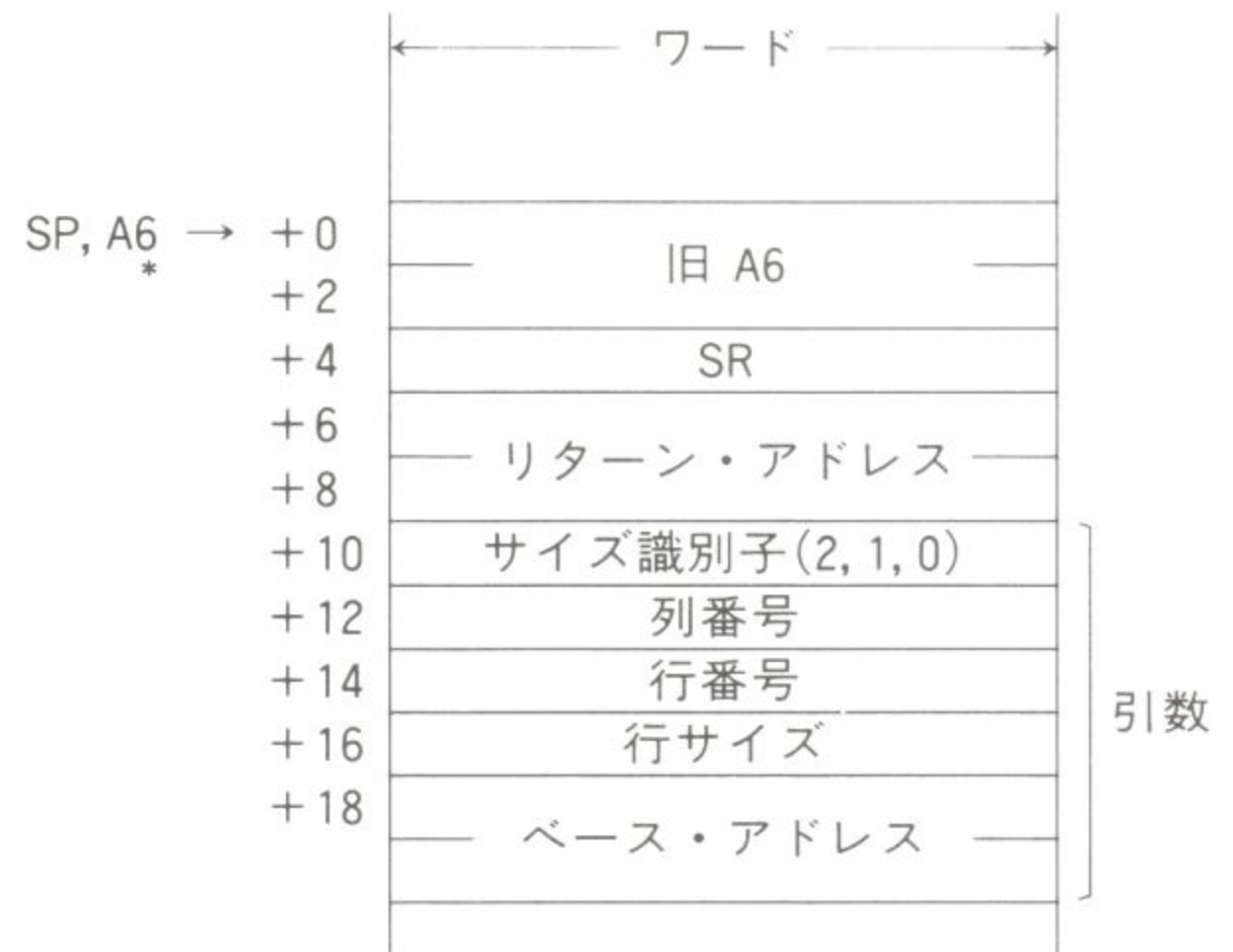
アセンブラの場合にはCコンパイラと異なり，かなりの部分まで最適化可能である．

本例では行間オフセットを計算によって求めたが，メモリ空間に余裕があればあらかじめ計算しておくことも許され，これらのアドレスをテーブル・インデックス方式で求めることが可能である．

この様子は3次元以上の配列にも同様に適用でき，より巨大な配列を広大なメモリ空間上でアクセスする場合には，Cコンパイラに格段の差をつけることになる．つまり，アセンブラではパラメータを〝決めつける〟ことができ，そのために余分な条件分岐やパラメータの計算が不要になるのである．

算し，そのアドレスを引数エリアへ返している．48行で列番号をクリアしているが，これはD2の上位ワードを$0000としたいためである．つまり，MULUの結果はロング・ワードなので，最後にアドレスの加算を行う場合に不便だからである．

行70〜92：7行6列の配列をロング・ワード，ワード，バイトで定義している．

図2.39 OBT__ARRAYとスタック

●行のサイズ＝(列の個数)×(データ・サイズ)
●行の先頭オフセット＝(行番号)×(行のサイズ)
●列のオフセット＝(列番号)×(データ・サイズ)
●配列のアドレス＝(配列のベース・アドレス)＋(行の先頭オフセット)＋(列のオフセット)

リスト［ARRAY］

```
LINE ADDR. CODE/VALUE          LABEL    OP      OPERAND        COMMENT

0005
0006        =00000002          L_POWER  EQU     2
0007        =00000001          W_POWER  EQU     1
0008        =00000000          B_POWER  EQU     0
0009
0010        =00000006          ROW_NUM  EQU     6
0011        =00000018          L_CSIZE  EQU     ROW_NUM*4      ;column size of long word
0012        =0000000C          W_CSIZE  EQU     ROW_NUM*2      ;column size of word
0013        =00000006          B_CSIZE  EQU     ROW_NUM        ;column size of byte
0014        =00000006
0016        =00005000                   ORG     $5000
0017                           *
0018                           *      entry    D0 = column size
0019                           *               D1 = column number
0020                           *               D2 = row number
0021                           *               D3 = power of 2
0022                           *
0023                           *               A0 = array base address
0024
0025        =00005000          ARRAY
0026 005000 2F08                        MOVE.L  A0,-(SP)       ;push base address
0027 005002 3F00                        MOVE.W  D0,-(SP)       ;push column size
0028 005004 3F01                        MOVE.W  D1,-(SP)       ;push column number
0029 005006 3F02                        MOVE.W  D2,-(SP)       ;push row number
0030 005008 3F03                        MOVE.W  D3,-(SP)       ;push power of 2
0031 00500A 4EB9 0000 501E              JSR     OBT_ARY
0032
0033 005010 DFFC 0000 000C              ADDA.L  #12,SP         ;adjust stack pointer
0034 005016 206F FFFC                   MOVE.L  -4(SP),A0      ;result
0035 00501A                    BREAK_0
0037 00501A 7000                        MOVEQ   #0,D0
0038 00501C 4E40                        TRAP    #0
0039
```

```
0040                          *               ---------------------------
0041                          *               [sub]:obtaine array address
0042                          *               ---------------------------
0043 00501E                   OBT_ARY
0044 00501E 40E7                      MOVE.W  SR,-(SP)
0045 005020 4E56 0000                 LINK    A6,#0
0046 005024 48E7 F080                 MOVEM.L D0-D3/A0,-(SP)
0047
0048 005028 7400                      MOVEQ   #0,D2           ;clear D2(row number)
0049 00502A 206E 0012                 MOVE.L  18(A6),A0       ;A0: array base address
0050 00502E 302E 0010                 MOVE.W  16(A6),D0       ;D0: column size
0051 005032 322E 000E                 MOVE.W  14(A6),D1       ;D1: column number
0052 005036 342E 000C                 MOVE.W  12(A6),D2       ;D2: row number
0053 00503A 362E 000A                 MOVE.W  10(A6),D3       ;D3: power of 2
0054
0055 00503E 6702                      BEQ     S_ARRAY
0056 005040 E7AA                      LSL.L   D3,D2           ;D2=(4,2)*(row number)
0057 005042                   S_ARRAY
0058 005042 C2C0                      MULU    D0,D1           ;D1= (column size)*(column number)
0059 005044 D481                      ADD.L   D1,D2           ;D2= offset
0060 005046 41F0 2800                 LEA     0(A0,D2.L),A0   ;obtaine array address
0061 00504A 2D48 0012                 MOVE.L  A0,18(A6)       ;load array address to argument area
0062
0063 00504E 4CDF 010F                 MOVEM.L (SP)+,D0-D3/A0
0064 005052 4E5E                      UNLK    A6
0065 005054 4E77                      RTR
0066
0067                          *               ---------------------------------
0068                          *               array(CLM,ROW) /* test data area
0069                          *               ---------------------------------
0070 005056 0000 000A 0000    L_ARRAY DC.L    10,11,12,13,14,15
            000B 0000 000C
            0000 000D 0000
            000E 0000 000F
0071 00506E 0000 0010 0000            DC.L    16,17,18,19,20,21
            0011 0000 0012
            0000 0013 0000
            0014 0000 0015
0072 005086 0000 0016 0000            DC.L    22,23,24,25,26,27
            0017 0000 0018
            0000 0019 0000
            001A 0000 001B
0073 00509E 0000 001C 0000            DC.L    28,29,30,31,32,33
            001D 0000 001E
            0000 001F 0000
            0020 0000 0021
0074 0050B6 0000 0022 0000            DC.L    34,35,36,37,38,39
            0023 0000 0024
            0000 0025 0000
            0026 0000 0027
0075 0050CE 0000 0028 0000            DC.L    40,41,42,43,44,45
            0029 0000 002A
            0000 002B 0000
            002C 0000 002D
0076 0050E6 0000 002E 0000            DC.L    46,47,48,49,50,51
            002F 0000 0030
            0000 0031 0000
            0032 0000 0033
0077
0078 0050FE 000A 000B 000C    W_ARRAY DC.W    10,11,12,13,14,15
            000D 000E 000F
0079 00510A 0010 0011 0012            DC.W    16,17,18,19,20,21
            0013 0014 0015
0080 005116 0016 0017 0018            DC.W    22,23,24,25,26,27
            0019 001A 001B
0081 005122 001C 001D 001E            DC.W    28,29,30,31,32,33
            001F 0020 0021
0082 00512E 0022 0023 0024            DC.W    34,35,36,37,38,39
            0025 0026 0027
0083 00513A 0028 0029 002A            DC.W    40,41,42,43,44,45
            002B 002C 002D
0084 005146 002E 002F 0030            DC.W    46,47,48,49,50,51
            0031 0032 0033
0085
0086 005152 0A0B 0C0D 0E0F    B_ARRAY DC.B    10,11,12,13,14,15
0087 005158 1011 1213 1415            DC.B    16,17,18,19,20,21
0088 00515E 1617 1819 1A1B            DC.B    22,23,24,25,26,27
0089 005164 1C1D 1E1F 2021            DC.B    28,29,30,31,32,33
0090 00516A 2223 2425 2627            DC.B    34,35,36,37,38,39
0091 005170 2829 2A2B 2C2D            DC.B    40,41,42,43,44,45
0092 005176 2E2F 3031 3233            DC.B    46,47,48,49,50,51
0093
0094        =00005000                 END     ARRAY
```

第2部 プログラム

15 文字列の扱い方

ここでは文字列処理用にいくつかのサブルーチンを作成しますが，現場での要求に応じて対処しなければなりません．

文字列操作のもつ意味ですが，たとえば，Cコンパイラやアセンブラなどの〝言語〟を開発する際には，それらのソース・プログラムは一連の文字列としてコンピュータへ入力されますから，様々な文字列操作が要求されることになります．そこで，いきなり「このサブルーチンは文字列の長さを求めるものです」などと説明されても，現場でそのような処理が要求されないのであれば，やはりピンとこないのは当然です．

■文字列自身のもつ意味

我々が表現するものはすべて文字であり，たとえば，255は〝にひゃく　ごじゅう　ご〟であるわけですが，コンピュータへは単なる3文字として入力され，コンピュータ内部で演算可能な形式に変換されます．同様に，

```
printf("68000 asm¥n");
MOVEA.L A0,A1
```

などにしても，文字列としてコンピュータ内部へ読み込まれ，コンパイルやアセンブルという処理がなされ，われわれの期待している結果が得られることになります．

■文字列とその記憶形式

数値表現ならメモリ・サイズは即座に求められますが，文字列の場合は〝長さ〟が不定であるため，メモリ効率はきわめて低下します．

文字列をどのようにメモリ内へ記憶するかは，プログラマ自身に委ねられるわけですが，Cコンパイラのように$00でターミネートしたり，文字列の長さと記憶されている先頭アドレスから成るテーブル（表）を作成してもよいでしょう（特に後者は文字列のソートには不可欠である）．

1 文字列の長さを求める

◉サンプルプログラム ［LEN］

文字列の長さを求めますが，ここでは終了地点を指定できるようにしました．

■動作

サブルーチンS_LENへ文字列の先頭アドレスと終了コードを渡すと，文字列の長さを返してくる．

■解法

メインから渡された文字列の終了コードが見つかるまで，１字１字カウントすればよい．

■各行の意味

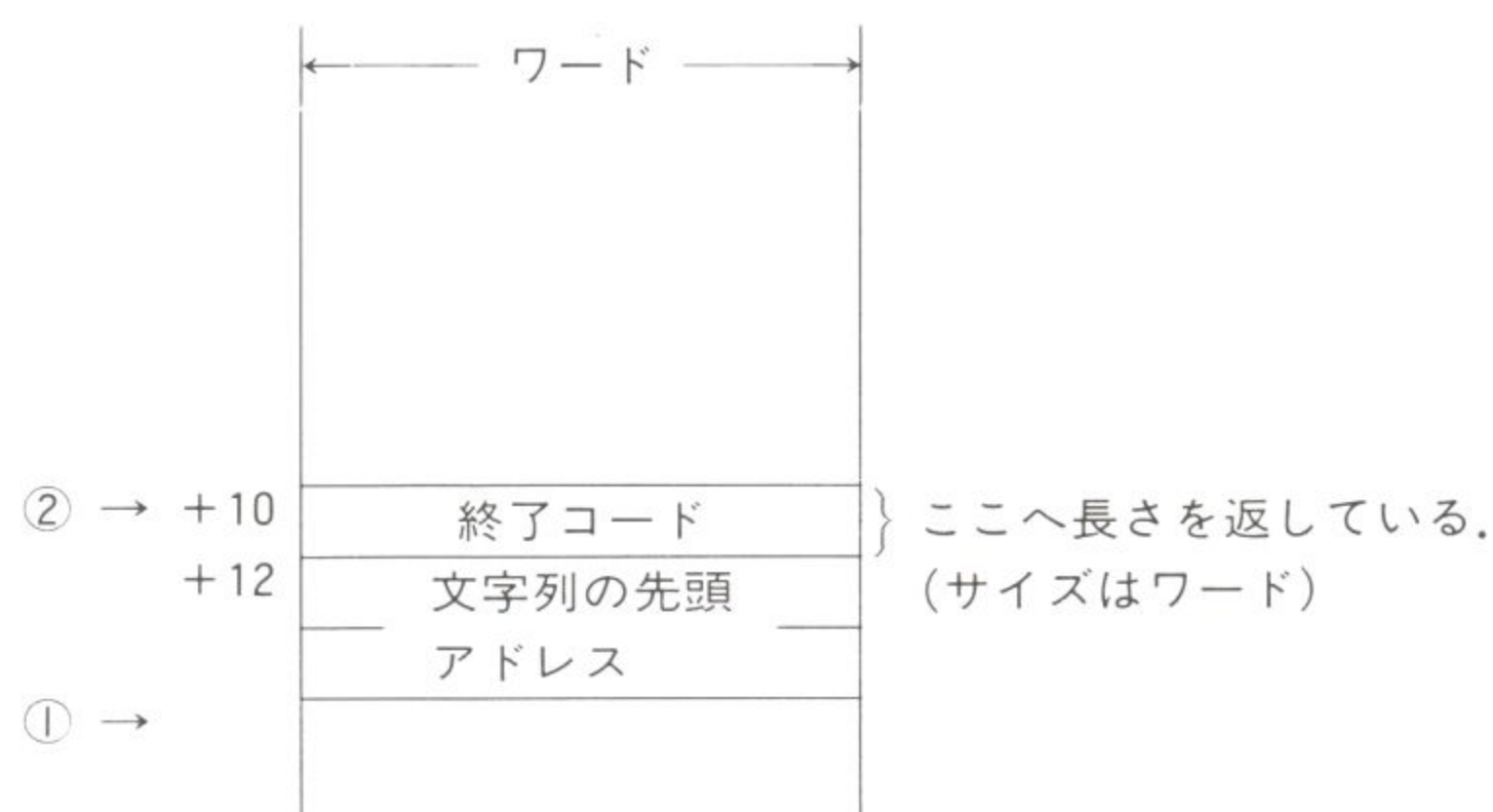
行 9～11：文字列の先頭アドレスと，D0に格納してある文字列の終了コードをスタックへプッシュし，S_LENを呼び出す．

行13～14：S_LENでは終了コードの位置に文字列の長さを返すので，スタックを復元し，引数を取り出している．
終了コードの次のアドレスを知りたければLENの仕様を変更し，アドレスも返すようにすればよく，その場合は，

```
MOVE.W    (SP)+,D1     :D1に長さを得る
MOVEA.L   (SP)+,A1     :A1にアドレスを得る
```

のようにして結果を受け取る．

行23～41：サブルーチン

図2.40　LENとスタック

①メイン側で引数をプッシュしない状態でのSPはここをポイントしている．
②LINK実行後A6からのディスプレースメントを示す．

リスト[LEN]

```
LINE ADDR.  CODE/VALUE          LABEL    OP        OPERAND          COMMENT

0005
0007        =00005000                    ORG       $5000
0008        =00005000           LEN
0009 005000 4879 0000 5040               PEA       STR              ;push string address
0010 005006 3F00                         MOVE.W    D0,-(SP)         ;push delimita
0011 005008 4EB9 0000 5018               JSR       S_LEN
0012
0013 00500E 5C8F                         ADDQ.L    #2*3,SP          ;adjust stack pointer
0014 005010 322F FFFA                    MOVE.W    -6(SP),D1        ;result
0015 005014                     BREAK
0017 005014 7000                         MOVEQ     #0,D0
0018 005016 4E40                         TRAP      #0
0019 005018
0020                            *                 -----------------------
0021                            *                 [sub] get string length
0022                            *                 -----------------------
0023 005018                     S_LEN
0024 005018 40E7                         MOVE.W    SR,-(SP)
0025 00501A 4E56 0000                    LINK      A6,#0
0026 00501E 48E7 C080                    MOVEM.L   D0-D1/A0,-(SP)
0027
0028 005022 4241                         CLR.W     D1               ;clear string counter
0029 005024 206E 000C                    MOVEA.L   12(A6),A0        ;A0: address
0030 005028 302E 000A                    MOVE.W    10(A6),D0        ;D0: delimita
0031 00502C                     SLEN_LP
0032 00502C B018                         CMP.B     (A0)+,D0         ;end of string ?
0033 00502E 6704                         BEQ       STR_END
0034 005030 5241                         ADDQ.W    #1,D1            ;countup
0035 005032 60F8                         BRA       SLEN_LP
0036 005034                     STR_END
0037 005034 3D41 000A                    MOVE.W    D1,10(A6)
0038
0039 005038 4CDF 0103                    MOVEM.L   (SP)+,D0-D1/A0
0040 00503C 4E5E                         UNLK      A6
0041 00503E 4E77                         RTR
0042
0043                            *                 --------------
0044                            *                 test data area
0045                            *                 --------------
0046 005040 6162 6364 6566      STR      DC.B      "abcdef",0
            00
0047
0048        =00005000                    END       LEN
```

2 $00で終了する文字列の長さを求める

◉サンプルプログラム ［LENZ］

文字列の長さを求めますがここでは文字列は$00で終了することになっているので，先のように終了コードを渡すことは不要です．

■動作

文字列の格納されている先頭アドレスをサブルーチンS_LENZへ渡し，文字列の長さを得るが，メイン側でのスタックの深さを考慮し，長さはロング・ワードとする．

■解法

$00が見つかるまで１字１字カウントすればよい．

■各行の意味

行 9 〜12：メイン側の処理

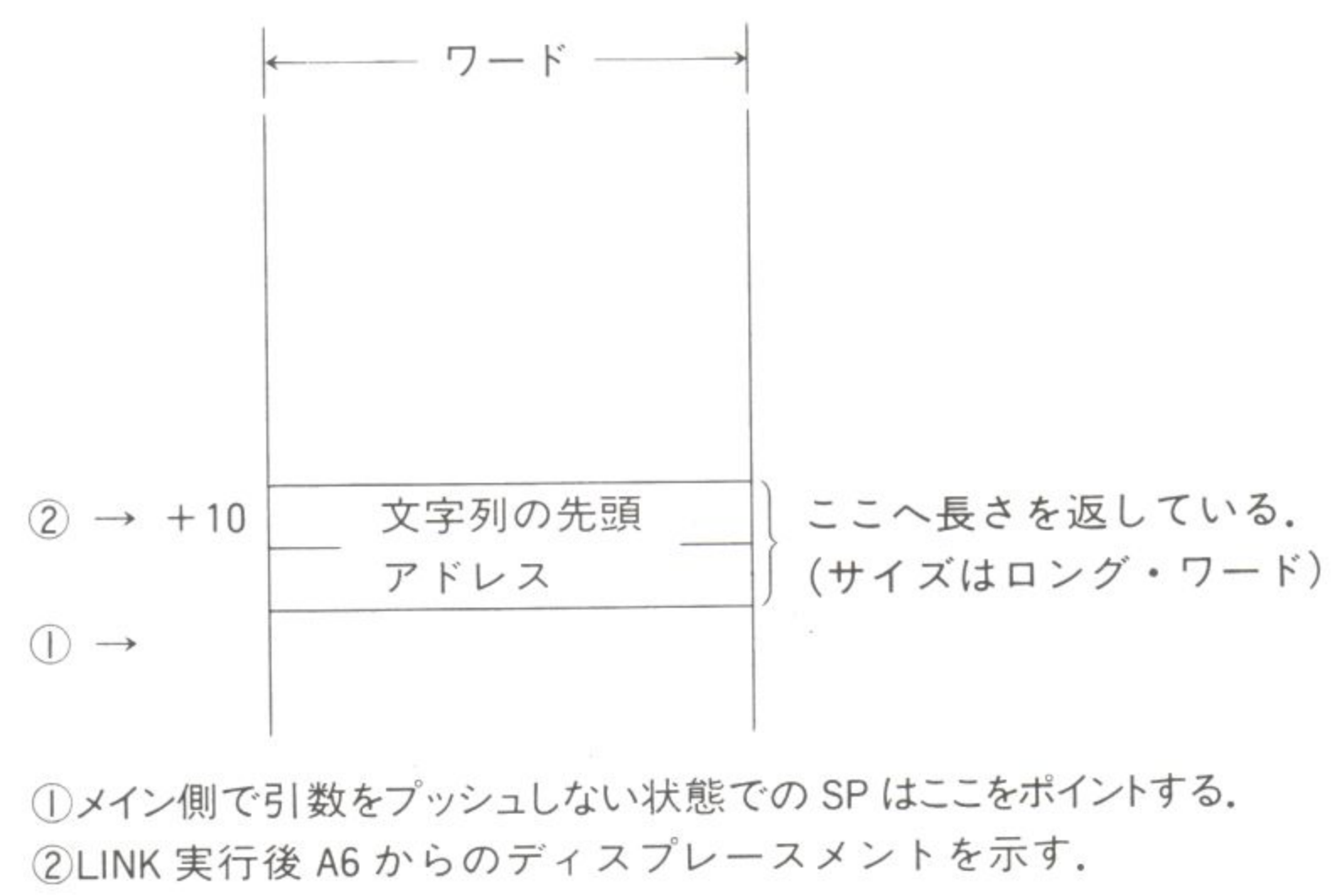
行22〜38：文字列の先頭アドレスを受け取り，$00がみつかるまでの文字数をカウントし，その値（ロング・ワード）を返す．

アプリケーション・ヒント

$00の次の文字が格納されているアドレスを受け取りたい場合，メイン側でそのためのスタック領域を確保するために余分にロング・ワードをプッシュするか，強制的にプッシュ動作を実行したようにスタックを操作し，それからS_LENZを呼び出す．

図2.41　LENZとスタック

ワード
② → ＋10　文字列の先頭アドレス　ここへ長さを返している．（サイズはロング・ワード）
① →

①メイン側で引数をプッシュしない状態でのSPはここをポイントする．
②LINK実行後A6からのディスプレースメントを示す．

リスト[LENZ]

```
LINE ADDR. CODE/VALUE          LABEL     OP       OPERAND          COMMENT

0005
0007        =00005000                    ORG      $5000
0008        =00005000          LENZ
0009 005000 4879 0000 5036               PEA      STR              ;push string address
0010 005006 4EB9 0000 5012               JSR      S_LENZ
0011
0012 00500C 221F                         MOVE.L   (SP)+,D1         ;result
0013 00500E                    BREAK
0015 00500E 7000                         MOVEQ    #0,D0
0016 005010 4E40                         TRAP     #0
0017 005012
0018                           *                -----------------------
0019                           *                [sub] get string length
0020                           *                -----------------------
0021 005012                    S_LENZ
0022 005012 40E7                         MOVE.W   SR,-(SP)
0023 005014 4E56 0000                    LINK     A6,#0
0024 005018 48E7 C080                    MOVEM.L  D0-D1/A0,-(SP)
0025
0026 00501C 4280                         CLR.L    D0               ;clear string counter
0027 00501E 206E 000A                    MOVEA.L  10(A6),A0        ;A0: address
0028 005022                    SLEN_LP
0029 005022 1218                         MOVE.B   (A0)+,D1
0030 005024 6704                         BEQ      STR_END
0031 005026 5280                         ADDQ.L   #1,D0            ;countup D0
0032 005028 60F8                         BRA      SLEN_LP
0033 00502A                    STR_END
0034 00502A 2D40 000A                    MOVE.L   D0,10(A6)        ;return length
0035
0036 00502E 4CDF 0103                    MOVEM.L  (SP)+,D0-D1/A0
0037 005032 4E5E                         UNLK     A6
0038 005034 4E77                         RTR
0039
0040                           *                --------------
0041                           *                test data area
0042                           *                --------------
0043 005036 6162 6364 6566     STR       DC.B     "abcdef",0
            00
0044
0045        =00005000                    END      LENZ
```

3 $00で終了する文字列中のポジションを求める

◉サンプルプログラム ［POS］

文字列の中から特定文字の位置を求めるもので，比較的よく利用されるものです．

たとえば，メニューに応じた処理ルーチンへ分岐する際のインデックスや，バイナリ～16進変換にも利用できます．

■動作

引数として文字列の先頭アドレスとサーチ対象となる文字コードをS_POSZへ渡す．

結果は次のようになる．

・指定文字が文字列中に存在する　：文字列の先頭位置を$00としたときの位置(ワード)と，文字の格納されているアドレスを得る．

・指定文字が文字列中に存在しない：$FFFFが得られる．

■解法

① 文字列は$00で終了する約束であるから，$00を取り出したら作業を終了するが，指定文字が存在しないことをメインへ知らせるために，文字位置として$FFFFを返す．

② 比較ループへ入る前に位置カウンタをクリアし，とにかく$00を見つけるまで1字1字指定文字と比較し，一致するまでカウンタを進める．カウンタを先に進めてから一致／不一致を判定せざるを得ないので，一致した場合のポジション・カウンタは1つだけ多くなる．

比較の終了条件は次の2つである．

- **$00を見つけたとき**
- **指定文字が見つかったとき**

■各行の意味

行11～16：最初に文字列の先頭アドレス，次に知りたい文字コードをワード・サイズでスタックへプッシュし，サブルーチンを呼び出し，結果を受け取っている．

行26～51：サブルーチン部．比較処理の終了条件，ポジション・カウンタの進め方，などに注意していただきたい．

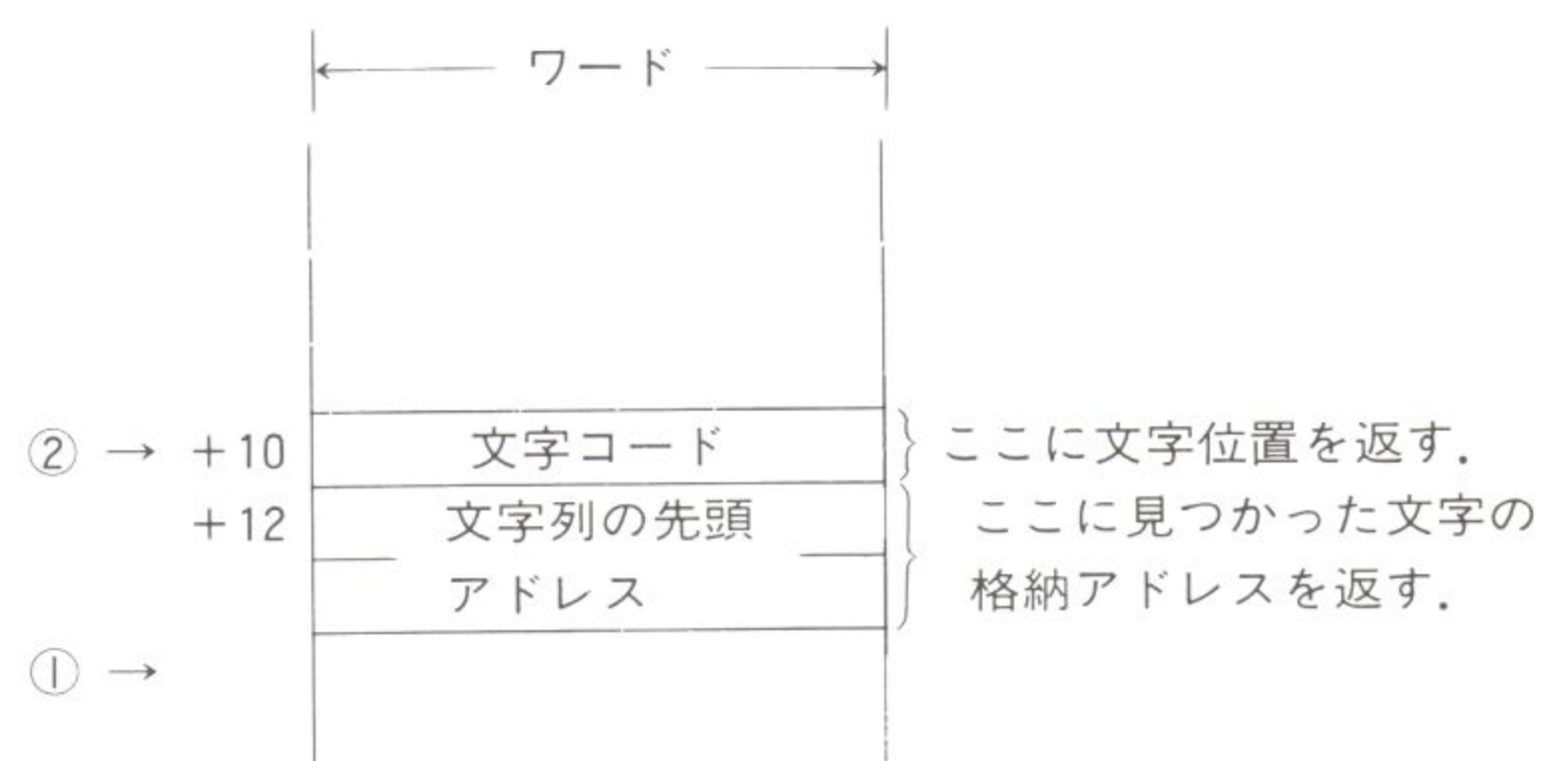
アプリケーション・ヒント

先にちょっとふれた16進の扱いであるが，キーボードからは文字としての`a´や`f´が入力されるから，内部演算には$A，$Fとする必要がある．そこで，

`0123456789abcdef´

`0123456789ABCDEF´

のような文字列を用意しておき，`a´の位置をサーチすれば10(じゅう)，すなわち$Aという数値が容易に得られる．2つの文字列セットを用意することで，大文字／小文字の区別なく変換できる．

図2.42　POSとスタック

①メイン側で引数をプッシュしない状態での SP はここをポイントしている．

②LINK 実行後 A6 からのディスプレースメントを示す．

リスト[POS]

```
LINE ADDR.  CODE/VALUE          LABEL    OP        OPERAND          COMMENT

0005
0006          =0000005A         T_STR    EQU       'Z'
0007
0009          =00005000                  ORG       $5000
0010          =00005000         POSZ
0011 005000 4879 0000 504E               PEA       STR              ;push string address pointer
0012 005006 3F3C 005A                    MOVE.W    #T_STR,-(SP)     ;push search character
0013 00500A 4EB9 0000 5018               JSR       S_POSZ
0014
0015 005010 321F                         MOVE.W    (SP)+,D1         ;result(position)
0016 005012 225F                         MOVEA.L   (SP)+,A1         ;result(address)
0017 005014                     BREAK
0019 005014 7000                         MOVEQ     #0,D0
0020 005016 4E40                         TRAP      #0
0021
0022                            *                 ---------------------
0023                            *                 [sub] search position
0024                            *                 ---------------------
0025 005018                     S_POSZ
0026 005018 40E7                         MOVE.W    SR,-(SP)
0027 00501A 4E56 0000                    LINK      A6,#0
0028 00501E 48E7 E080                    MOVEM.L   D0-D2/A0,-(SP)
0029
0030 005022 4240                         CLR.W     D0               ;clear pos_counter
0031 005024 322E 000A                    MOVE.W    10(A6),D1
0032 005028 206E 000C                    MOVEA.L   12(A6),A0
0033 00502C                     S_POSZL
0034 00502C 1418                         MOVE.B    (A0)+,D2         ;D2 Equal to Zero ?
0035 00502E 6604                         BNE       POS_NEXT
0036 005030 70FF                         MOVEQ     #$FF,D0          ;not found
0037 005032 6008                         BRA       ESC_POS
0038 005034                     POS_NEXT
0039 005034 5240                         ADDQ.W    #1,D0            ;pos_counter = pos_counter+1
0040 005036 B401                         CMP.B     D1,D2
0041 005038 66F2                         BNE       S_POSZL          ;next character
0042
0043 00503A 5340                         SUBQ.W    #1,D0            ;format position
0044 00503C                     ESC_POS
0045 00503C 5388                         SUBQ.L    #1,A0            ;format string address pointer
0046 00503E 3D40 000A                    MOVE.W    D0,10(A6)        ;return position
0047 005042 2D48 000C                    MOVE.L    A0,12(A6)        ;return string address pointer
0048
0049 005046 4CDF 0107                    MOVEM.L   (SP)+,D0-D2/A0
0050 00504A 4E5E                         UNLK      A6
0051 00504C 4E77                         RTR
0052
0053                            *                 --------------
0054                            *                 test data area
0055                            *                 --------------
0056 00504E 6162 6364 655A      STR      DC.B      "abcdeZ",0
            00
0057
0058          =00005000                  END       POSZ
```

4 ブランクの読み飛ばし

◉サンプルプログラム[S_SPC]

スペース・コード（$20）やタブ・コード（$9）を読み飛ばし，次に位置する〝意味のある文字〟が格納されているアドレスを返すサブルーチンです.

■動作

文字列の先頭アドレスをサブルーチンSKIP_SPCへ渡し，結果としてスペースまたはタブの次のアドレスを得る.

■解法

① 文字列から1文字読み取り，スペース・コードまたはタブ・コードを見つける.

② 次にスペース・コードまたはタブ・コードのいずれにも該当しない文字を見つける. このようにする理由は，これらの文字コードが1つであるとは限らないからである.

■各行の意味

行12〜14：メイン・ルーチン

行24〜43：サブルーチン.
スペース・コードやタブ・コードをデータ・レジスタへ格納してから比較命令を実行しているが，イミディエート・モードで比較してもよい.

行31〜37：1個目のスペースまたはタブをサーチする部分である.

行38〜43：有効文字が見つかるまでスペースまたはタブを読み飛ばす.

アプリケーション・ヒント

本例では文字列の終了コードをチェックしていないが，特に文（コンパイラやアセンブラ）の解釈には必要である.

リスト[S_SPC]

```
LINE ADDR. CODE/VALUE          LABEL     OP       OPERAND          COMMENT

0005
0006        =00000020          SPC_CODE EQU      $20
0007        =00000009          TAB_CODE EQU      9
0008
0010        =00005000                   ORG      $5000
0011        =00005000          S_SPC
0012 005000 2F08                        MOVE.L   A0,-(SP)         ;A0: string pointer
0013 005002 4EB9 0000 500E              JSR      SKIP_SPC
0014 005008 225F                        MOVEA.L  (SP)+,A1         ;A1: result
0015 00500A                    BREAK
0017 00500A 7000                        MOVEQ    #0,D0
0018 00500C 4E40                        TRAP     #0
0019
0020                           *        ----------------------------
0021                           *        [sub] sikp tab or space code
0022                           *        ----------------------------
0023 00500E                    SKIP_SPC
```

```
0024 00500E 40E7                       MOVE.W   SR,-(SP)
0025 005010 4E56 0000                  LINK     A6,#0
0026 005014 48E7 E080                  MOVEM.L  D0-D2/A0,-(SP)
0027
0028 005018 206E 000A                  MOVEA.L  10(A6),A0
0029 00501C 123C 0020                  MOVE.B   #SPC_CODE,D1
0030 005020 143C 0009                  MOVE.B   #TAB_CODE,D2
0031 005024                  SPC_LOOP
0032 005024 1018                       MOVE.B   (A0)+,D0
0033 005026 B001                       CMP.B    D1,D0          ;space code ?
0034 005028 6706                       BEQ      SPC_LP2
0035 00502A B002                       CMP.B    D2,D0          ;tab code ?
0036 00502C 6702                       BEQ      SPC_LP2
0037 00502E 60F4                       BRA      SPC_LOOP       ;next character
0038 005030                  SPC_LP2
0039 005030 1018                       MOVE.B   (A0)+,D0
0040 005032 B001                       CMP.B    D1,D0          ;space code ?
0041 005034 67FA                       BEQ      SPC_LP2
0042 005036 B002                       CMP.B    D2,D0          ;tab code ?
0043 005038 67F6                       BEQ      SPC_LP2
0044
0045 00503A 5388                       SUBQ.L   #1,A0
0046 00503C 2D48 000A                  MOVE.L   A0,10(A6)      ;return next address
0047
0048 005040 4CDF 0107                  MOVEM.L  (SP)+,D0-D2/A0
0049 005044 4E5E                       UNLK     A6
0050 005046 4E77                       RTR
0051
0052                         *                 ---------
0053                         *                 data area
0054                         *                 ---------
0055 005048 6162 6364 65     SDATA_A   DC.B     "abcde"
0056 00504D 20                        DC.B     SPC_CODE
0057 00504E 6162 63                    DC.B     "abc"
0058
0059 005051 6162 6364 65     SDATA_B   DC.B     "abcde"
0060 005056 0909                       DC.B     TAB_CODE,TAB_CODE
0061 005058 6162 63                    DC.B     "abc"
0062
0063 00505B 2020 20          SDATA_C   DC.B     SPC_CODE,SPC_CODE,SPC_CODE
0064 00505E 7374 7269 6E67             DC.B     "string_c"
            5F63
0065
0066 005066 09               SDATA_D   DC.B     TAB_CODE
0067 005067 7374 7269 6E67             DC.B     "string_d"
            5F64
0068
0069        =00005000                  END      S_SPC
```

5 $00で終了する文字列の管理テーブルを作成

◉サンプルプロプログラム ［FSPNT］

文字列を管理するには〝長さ〟と〝1文字目の格納アドレス〟の2つの情報が必要であり，本例では順に文字列の長さ（ワード），次に格納アドレス（ロング・ワード）というフォーマットを想定し，1つの文字列を管理するために6バイトを割り当てます．

■動作

メイン側からは文字列の個数，最初の文字列格納アドレス，管理テーブルの先頭アドレスをスタックへプッシュしFM_SPNTを呼び出し，文字列管理テーブルを作成する．

ただしサブルーチン側から返される情報はない．

■解法

文字列の格納状態であるが,$00で終了するものがメモリ上にギッシリ格納されているものとし，あらかじめ文字列の総数が判明しているものとした．場合によっては，各文字列がメモリ上にランダムに存在することもあるが，たとえば，ディスク上にランダムに存在する文字列をメモリ上へ読み込んだ状態を想定している．

まず$00を見つけるまでの文字数をカウントするが,先頭アドレスを別のレジスタへ保存しておく必要がある．いずれにしても強力なアドレッシング・モードと豊富なレジスタ群により，すんなり記述できる．

文字列の長さはこれまでのサブルーチンを利用して対処できるが,68000には豊富なレジスタがあること，また処理も複雑ではないので，本サブルーチン内で求めている．

■各行の意味

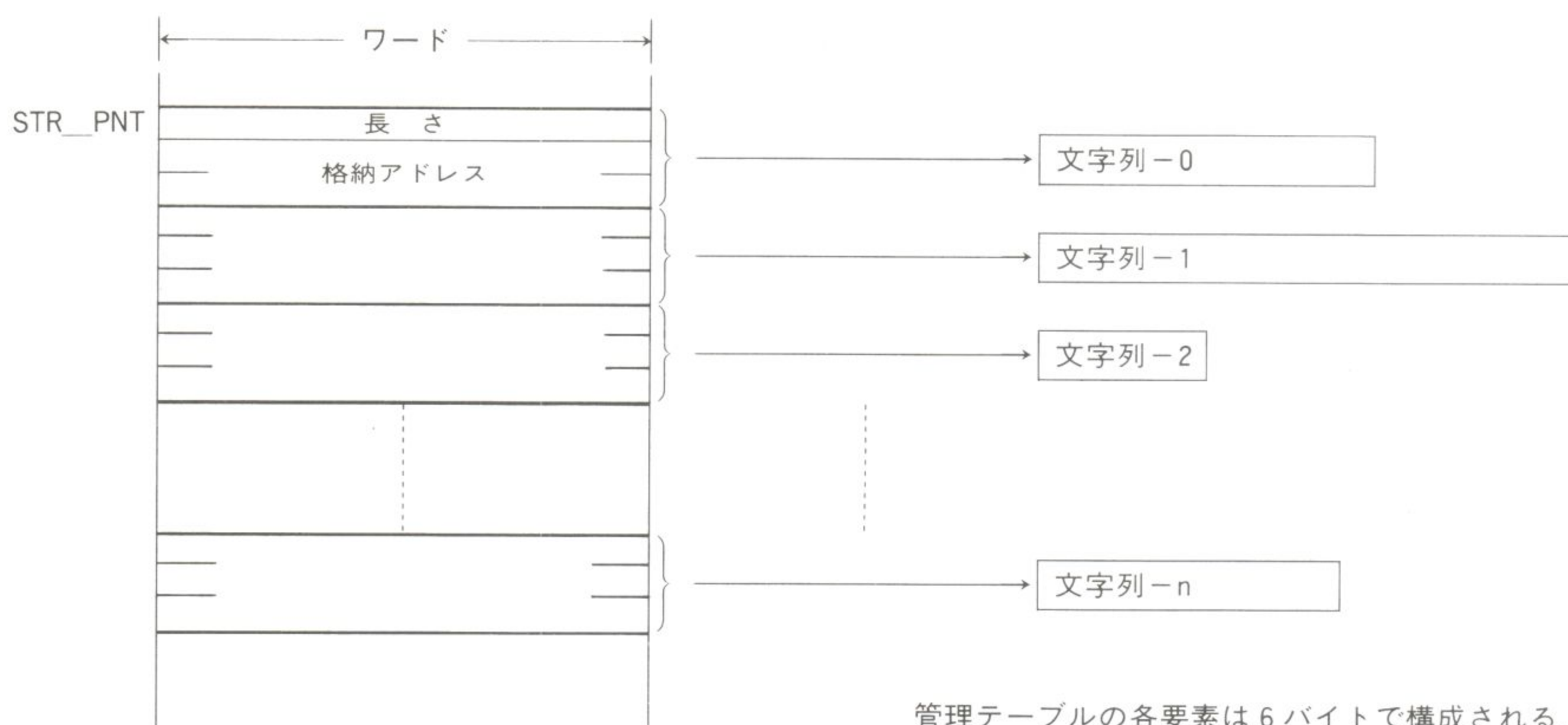
行11～16：3つの引数をスタックへプッシュし,FM__SPNTを呼び出す．16行では10バイトだけSPが更新されているので，これを復元している．

行25～48：文字列の管理表を作成するルーチンで，各レジスタは以下のようにアサインされる．

- **D0**　：全体のループ・カウンタ
- **D1**　：文字列カウンタ
- **D2**　：$00を見つけるための作業用
- **A0**　：文字列から1字1字読む際のポインタ
- **A1**　：管理テーブルへのポインタ
- **A2**　：文字列の先頭アドレスを保持する作業用

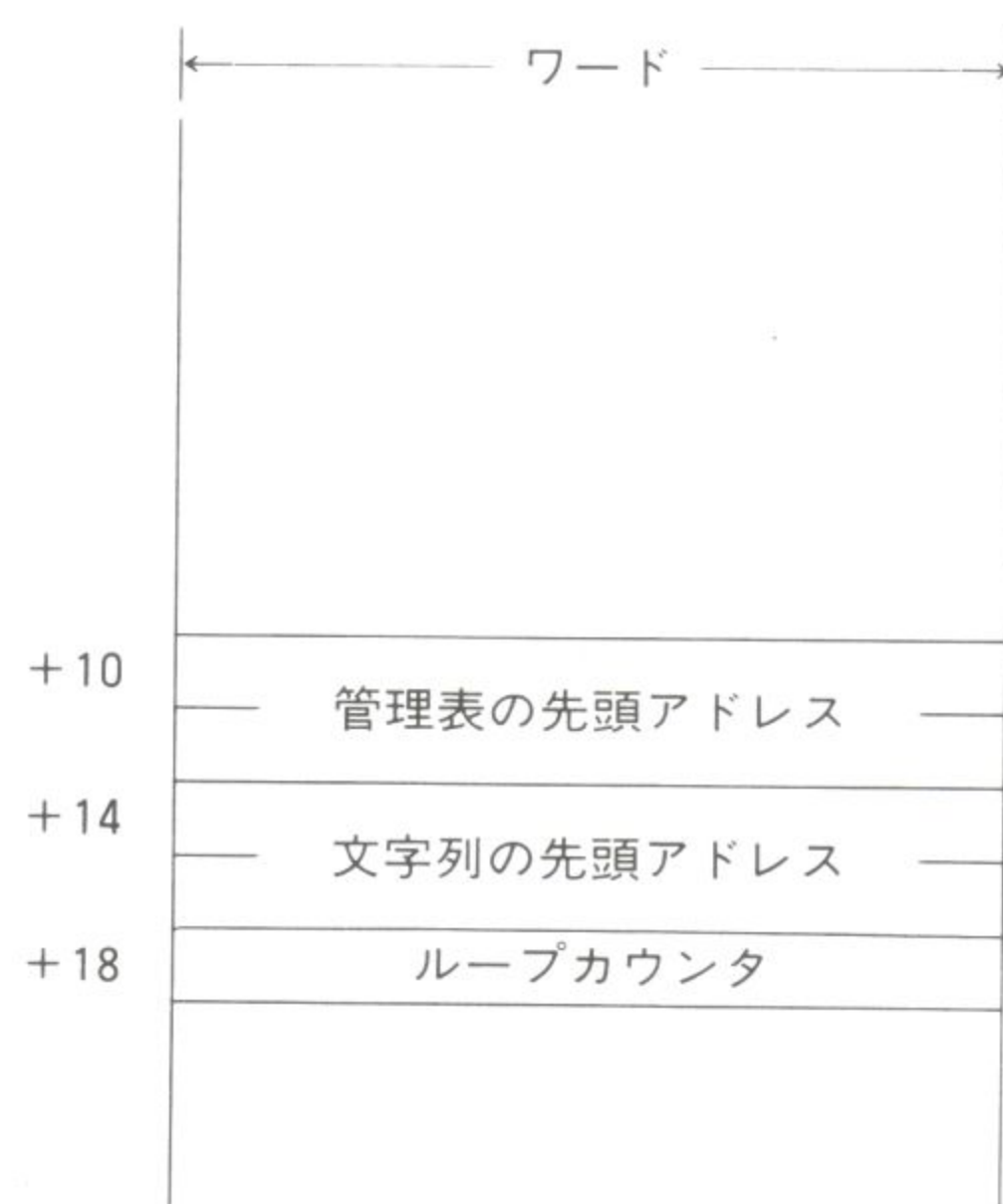
行62　：EVENという擬似命令が置かれているが，文字列はバイト単位で記憶されることから,STR__PNTが奇数ロケーションに置かれる心配がある．
一方STR__PNTはワードやロング・ワードでアクセスされるので,ロケーションは偶数でなければならない．このようなことから,STR__PNTを強制的に偶数アドレスへロケートするためにEVENを使用している．

文字列とその管理テーブルとの対応は**図2.43**の通りである．

図2.43　文字列と管理テーブル

アプリケーション・ヒント

実際には文字列がどこから入力されるか，それはどこで終了するか，という問題がある．たとえばディスケット上に点在している文字列を読み込む場合には，ファイルの終了や文字列の格納形式の2つの条件があり，これらの条件をたよりに$00でターミネートされる文字列を文字列領域へ順に格納する(この時に文字列をカウントすることもできる)．このような前処理を行っておけば，FM__SPNTは十分有効に働く．

文字列のソートをする場合には，本例のような文字列管理テーブルを作成することはきわめて有効である．ソートという作業は文字列を比較し，その結果に応じて文字列の交換をしなければならず，そのためには文字列のブロック転送を3回実行しなければならない．ところが，管理テーブルを作成しておけば，その内容（文字列長と格納アドレス）の各セットを交換するだけでよく，長々とブロック転送することは不要となる．

図2.44　FSRNTとスタック

数値＋10～＋18は，LINKによるA6からのディスプレースメントを意味する．

リスト［FSPNT］

```
LINE ADDR.  CODE/VALUE          LABEL    OP        OPERAND          COMMENT

0005
0006         =00000002          NUMBER   EQU       2
0007
0009         =00005000                   ORG       $5000
0010         =00005000          FSPNT
0011 005000 3F3C 0001                    MOVE.W    #NUMBER-1,-(SP)
0012 005004 4879 0000 5052               PEA       STR
0013 00500A 4879 0000 5060               PEA       STR_PNT
0014 005010 4EB9 0000 5020               JSR       FM_SPNT
0015
0016 005016 DFFC 0000 000A               ADDA.L    #10,SP           ;adjust stack pointer
0017 00501C                     BREAK
0019 00501C 7000                         MOVEQ     #0,D0
0020 00501E 4E40                         TRAP      #0
0021
0022                            *                -----------------------------
0023                            *                [sub] format string descriptor
0024                            *                -----------------------------
0025 005020                     FM_SPNT
0026 005020 40E7                         MOVE.W    SR,-(SP)
0027 005022 4E56 0000                    LINK      A6,#0
0028 005026 48E7 E0E0                    MOVEM.L   D0-D2/A0-A2,-(SP)
0029
0030 00502A 302E 0012                    MOVE.W    18(A6),D0        ;D0: loop counter
0031 00502E 206E 000E                    MOVEA.L   14(A6),A0        ;A0: read pointer
0032 005032 226E 000A                    MOVEA.L   10(A6),A1        ;A1: write pointer
0033 005036                     FM_SLOOP
0034 005036 4241                         CLR.W     D1               ;D1: character work counter
0035 005038 2448                         MOVEA.L   A0,A2            ;A2: pointer of 1st character
0036 00503A                     SCOUNT
0037 00503A 1418                         MOVE.B    (A0)+,D2
0038 00503C 6704                         BEQ       LD_SPNT
0039 00503E 5241                         ADDQ.W    #1,D1            ;countup character
0040 005040 60F8                         BRA       SCOUNT
0041 005042                     LD_SPNT
0042 005042 32C1                         MOVE.W    D1,(A1)+         ;format sting length
0043 005044 22CA                         MOVE.L    A2,(A1)+         ;format sting pointer
0044 005046 51C8 FFEE                    DBRA      D0,FM_SLOOP      ;end of sting ?
0045
0046 00504A 4CDF 0707                    MOVEM.L   (SP)+,D0-D2/A0-A2
0047 00504E 4E5E                         UNLK      A6
0048 005050 4E77                         RTR
0049
0050                            *                --------------
0051                            *                test data area
0052                            *                --------------
0053
0054 005052 6162 6364 6566      STR      DC.B      "abcdef",0
0055 005059 7677 7879 7A00               DC.B      "vwxyz",0
0056
0057                            *                -----------------------------------
0058                            *                data area for format string pointer
0059                            *                1st: string length (word)
0060                            *                2nd: string pointer(long word)
0061                            *                -----------------------------------
0062                                     EVEN
0063 005060 =0000000C           STR_PNT  DS.W      3*NUMBER
0064
0065         =00005000                   END       FSPNT
```

6 文字列の内容を交換する

◉サンプルプログラム [S__EXG]

$00で終了する文字列の格納アドレスを引数として渡し，文字列の内容を交換するサブルーチンです．

■動作

引数として2つの文字列格納アドレスをスタックへプッシュしEX_STRを呼ぶことで両文字列の内容を交換する．

■解法

文字列のコピーを3回実行するので，EX__STR内から呼び出す専用サブルーチンS__COPYを作成して対処しているが，交換という操作には作業用領域が必要であり，文字列の退避用に256バイトのローカル・エリアを確保している（$00までをコピーするための汎用ルーチンをEX__STRからコールしてもよいが，このための処理は単純であるのでこのようにした）．

また使用するアドレス・レジスタは再び使用することになるので，これらを保存しながらの処理になる．

文字列の交換は次のように行っている．

① A__STRの内容をローカル・エリアへ退避．

② A__STRがフリーになったので，B__STRの内容をA__STRへコピー．

③ B__STRがフリーになったので，ローカル・エリアに退避しておいた内容をA__STRへコピーする．

■各行の意味

行9～12：メイン・ルーチンである．

行22～44：文字列の交換をするサブルーチンである．

行28　：確保したローカル・エリアの最もゼロ番地よりのアドレスをA2に求めているが，23行でリンクされたSPは更新されているので，本行のようにA6をたよりに計算する．

行49～56：A3でポイントされるロケーションからA4でポイントされるロケーションへ，$00でターミネートされる文字列をコピーする．ここではレジスタやCCRの保存をまったく考慮していないが，このサブルーチンはEX__STRからのみ呼び出されることを想定しているので，メイン側からはすべてのレジスタが保存されているように機能する．

■注意事項

本例のように隣接して文字列が格納されている場合，長さが異なっているといずれかの文字列が破壊されるので，文字列を絶対アドレスで管理する場合の文字列交換は，文字列どうしの格納アドレスに十分な注意が必要である．

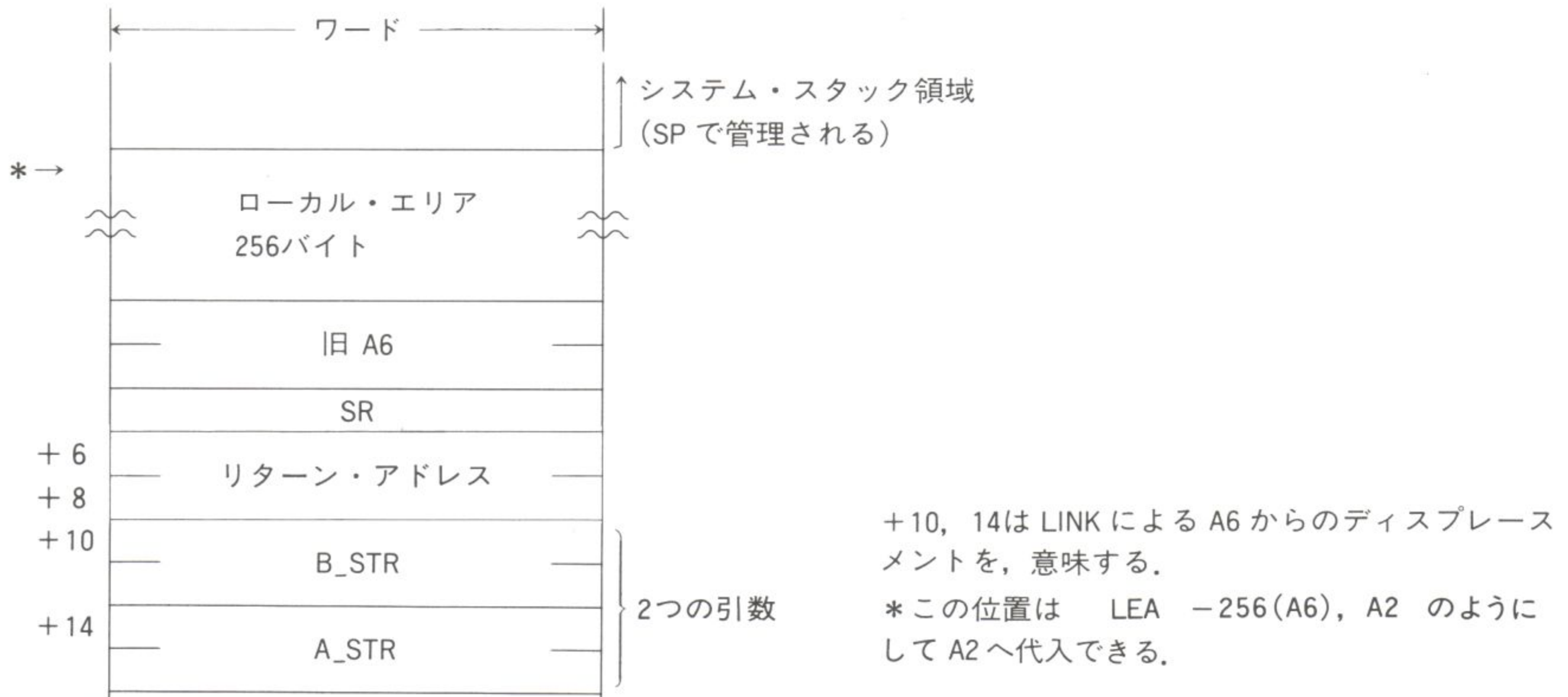
しかし現実には，必ず文字列が隣接して格納されるとは限らず，このような文字列の交換も時には要求されることがある．

アプリケーション・ヒント

文字列管理テーブルを作成し，この内容を参照して文字列を間接的に操作するならば，非常に柔軟な文字列操作を期待できる．たとえば，文字列の交換はテーブルの内容を交換するだけでよく，〝注意事項〟での文字列破壊の心配がないばかりか，処理スピードも格段にアップする．

図2.45 S_EXGとスタック

ワード
システム・スタック領域
(SPで管理される)
*→
ローカル・エリア
256バイト
旧A6
SR
+6
+8
リターン・アドレス
+10
B_STR
+14
A_STR
2つの引数

+10，14はLINKによるA6からのディスプレースメントを，意味する．
*この位置は LEA −256(A6)，A2 のようにしてA2へ代入できる．

リスト[S_EXG]

```
LINE ADDR. CODE/VALUE          LABEL    OP       OPERAND          COMMENT

0005
0007        =00005000                   ORG      $5000
0008        =00005000          S_EXG
0009 005000 4879 0000 5054              PEA      A_STR
0010 005006 4879 0000 5059              PEA      B_STR
0011 00500C 4EB9 0000 5018              JSR      EX_STR
0012 005012 508F                        ADDQ.L   #8,SP            ;adjust stack pointer
0013 005014                    BREAK
0015 005014 7000                        MOVEQ    #0,D0
0016 005016 4E40                        TRAP     #0
0017
0018                           *        ------------------------------
0019                           *        [sub] exchange a_str for b_str
0020                           *        ------------------------------
0021 005018                    EX_STR
0022 005018 40E7                        MOVE.W   SR,-(SP)
0023 00501A 4E56 FF00                   LINK     A6,#-256
0024 00501E 48E7 80F8                   MOVEM.L  D0/A0-A4,-(SP)
0025
0026 005022 206E 000E                   MOVEA.L  14(A6),A0        ;A0: a_str address pointer
0027 005026 226E 000A                   MOVEA.L  10(A6),A1        ;A1: b_str address pointer
0028 00502A 45EE FF00                   LEA      -256(A6),A2      ;A2: top local area
0029
0030 00502E 2648                        MOVEA.L  A0,A3            ;(1) copy a_str to local area
0031 005030 284A                        MOVEA.L  A2,A4
0032 005032 6114                        BSR      S_COPY
0033
0034 005034 2649                        MOVEA.L  A1,A3            ;(2) copy b_str to a_str
0035 005036 2848                        MOVEA.L  A0,A4
0036 005038 610E                        BSR      S_COPY
0037
0038 00503A 264A                        MOVEA.L  A2,A3            ;(3) copy local area to b_str
0039 00503C 2849                        MOVEA.L  A1,A4
0040 00503E 6108                        BSR      S_COPY
0041
0042 005040 4CDF 1F01                   MOVEM.L  (SP)+,D0/A0-A4
0043 005044 4E5E                        UNLK     A6
0044 005046 4E77                        RTR
0045
0046                           *        ------------------------------------
0047                           *        s_copy sub.  /* @(A3) --> @(A4) */
0048                           *        ------------------------------------
0049 005048                    S_COPY
0050 005048 101B                        MOVE.B   (A3)+,D0         ;@(A3) --> @(A4)
0051 00504A 6704                        BEQ      S_COPYE
0052 00504C 18C0                        MOVE.B   D0,(A4)+
0053 00504E 60F8                        BRA      S_COPY
0054 005050                    S_COPYE
0055 005050 4214                        CLR.B    (A4)             ;delimita (0)
0056 005052 4E75                        RTS
0057 005054
0058                           *        ---------
0059                           *        data area
0060                           *        ---------
0061 005054 6162 6364 00       A_STR    DC.B     "abcd",0
0062 005059 7778 797A 00       B_STR    DC.B     "wxyz",0
0063
0064        =00005000                   END      S_EXG
```

7 文字列の比較を行う

◉サンプルプログラム ［S_CMP］

文字列管理テーブルが作成されていることを前提とした文字列の比較を行ないます．

■動作

文字列管理テーブルのポインタをスタックへプッシュしSTR_CMPを呼び出すが，結果を直接CCRで受けるようにしている．サブルーチンを呼び出す直前のCCRは破壊されるが，その他のレジスタはこれまでの例のようにすべて保存される．

■解法

文字列比較のポイントは以下のようになる．

① 何文字比較するかは文字列の短い方の長さで決まる．

② 1文字の比較を所定回数実行するが，それでも大小を決定できない場合もありうる．この場合には文字列の長さの比較結果が文字列の大小関係となる．

③ 復帰情報は，大きい，小さい，等しい，の3通りである．

④ 文字列管理テーブルの意味は図2.43で取り上げたものと同様である．

■各行の意味

行12～14：引数をスタックへプッシュしSTR_CMPを呼び出す．A0，A1に格納される内容はSTR_TBL0～STR_TBL7であり，A0，A1の順にプッシュする（STR_CMPでは（A1）－（A0）のような演算を行って大小関係を求めている）．ここではスタックへプッシュされる内容とその順序が意味をもち，A0，A1にこだわる必要はない．

行16～17：スタックを回復する．16，17行の命令はA0，A1の内容を復帰するわけではなく，STR_CMPではCCR以外のすべてのレジスタを保存している．つまり，スタックを回復するのが目的だが，STR_CMPからの情報はCCRのみであるから，別にA0，A1を使用しなくてもよい．

または，アドレスレジスタへの加算命令はCCRを操作しないので，

```
ADDQ.L    #8,SP
```

または

```
ADDA.L    #8,SP
```

により，スタックを回復してもよい．

行27～28：おきまりのコースである．

行30～31：引数をA0，A1に受け取っている．このようにするならA0，A1の値そのものを渡せばよさそうなものだが，プログラム全体から見れば，A0やA1は他の用途に使われているかも知れない，ということを考えてほしい．というわけで，メイン側のA0，A1とSTR_CMP内でのA0，A1は完全に分離されているのである．

行33〜38：文字列管理テーブルの内容を参照し，いずれの文字数を採用するかを決定しているが，36行ではこのときの比較結果があとで必要になるため，CCRの内容をD2へ保存している．

行40〜42：文字列を１文字ずつ比較するためのループ値，アドレス・ポインタを初期化している．

行44〜47：実際の文字比較を行い結果をCCRへ求める．最後まで比較したが勝負がつかなかった場合は47行へ制御が移行し，ここでD2の内容をCCRへ代入している．

行49〜51：メインへの復帰処理部であるが，RTRではなくRTSでもどっている．

行56〜78：文字列管理テーブルが位置しているが，実際はこのようなテーブルを前処理として行うプログラムが必要となる．各テーブルの内容は文字列の長さ（ワード）と文字列の先頭格納アドレス（ロング・ワード）から構成される．ここのアドレスをメインからの引数としてSTR__CMPへ渡すのである．

行83〜93：文字列のサンプルを定義してあるが，たとえばSTR__0，STR__1とを比較した場合に得られるCCRの各ビットの状態をコメントとして明記したので，STR__CMPからの復帰情報をメイン側でどのように扱えばよいか理解できると思われる．

アプリケーション・ヒント

ここでの比較は１バイトの符号なし比較であるが，`A´と`a´では`a´の方が大きなコードなので辞書形式のような分類をするなら，すべて大文字に変換してからSTR__CMPを呼び出すか，STR__CMP内に同様な処理を行う部分が必要である．一般には，44行以降で比較処理を行っているので，作業用データ・レジスタに１文字読み込み，その内容が小文字であれば大文字に変換してから比較を行う（このようにしてもまだレジスタは余っている）．

リスト［S__CMP］

```
LINE ADDR. CODE/VALUE           LABEL     OP        OPERAND          COMMENT

0005
0007        =00005000                     ORG       $5000
0008        =00005000           S_CMP
0009                            *
0010                            *                  /* @(A1) - @(A0) */
0011                            *
0012 005000 2F08                          MOVE.L    A0,-(SP)         ;push 1st address pointer
0013 005002 2F09                          MOVE.L    A1,-(SP)         ;push 2nd address pointer
0014 005004 4EB9 0000 5012                JSR       STR_CMP
0015
0016 00500A 225F                          MOVEA.L   (SP)+,A1
0017 00500C 205F                          MOVEA.L   (SP)+,A0
0018 00500E                     BREAK
0020 00500E 7000                          MOVEQ     #0,D0
0021 005010 4E40                          TRAP      #0
0022
0023                            *                  --------------------
0024                            *                  [sub] string compare
0025                            *                  --------------------
0026 005012                     STR_CMP
0027 005012 4E56 0000                     LINK      A6,#0
```

```
0028 005016 48E7 E0C0                MOVEM.L D0-D2/A0-A1,-(SP)
0029
0030 00501A 206E 000C                MOVEA.L 12(A6),A0       ;load string pointer to A0
0031 00501E 226E 0008                MOVEA.L 8(A6),A1        ;load string pointer to A1
0032
0033 005022 3018                     MOVE.W  (A0)+,D0        ;/* select string length */
0034 005024 3219                     MOVE.W  (A1)+,D1
0035 005026 B240                     CMP.W   D0,D1           ;(D1)-(D0)
0036 005028 40C2                     MOVE.W  SR,D2           ;save CCR
0037 00502A 6402                     BCC     LD_SCNT         ;CC means (D1) >= (D0)
0038 00502C C141                     EXG     D0,D1           ;CS means (D1) <  (D0)
0039 00502E                LD_SCNT
0040 00502E 5340                     SUBQ.W  #1,D0           ;adjust loop counter
0041 005030 2050                     MOVEA.L (A0),A0         ;setup 1st string pointer
0042 005032 2251                     MOVEA.L (A1),A1         ;setup 2nd string pointer
0043 005034                L_CMPM
0044 005034 B308                     CMPM.B  (A0)+,(A1)+     ;compare loop
0045 005036 6606                     BNE     ESC_CMPM
0046 005038 51C8 FFFA                DBRA    D0,L_CMPM
0047 00503C 44C2                     MOVE.W  D2,CCR          ;return CCR
0048 00503E                ESC_CMPM
0049 00503E 4CDF 0307                MOVEM.L (SP)+,D0-D2/A0-A1
0050 005042 4E5E                     UNLK    A6
0051 005044 4E75                     RTS
0052
0053                       *         -----------------
0054                       *         string descriptor
0055                       *         -----------------
0056 005046 0005           STR_TBL0  DC.W    5               ;for STR_0
0057 005048 0000 5076                DC.L    STR_0
0058
0059 00504C 0005           STR_TBL1  DC.W    5               ;for STR_1
0060 00504E 0000 507B                DC.L    STR_1
0061
0062 005052 0006           STR_TBL2  DC.W    6               ;for STR_2
0063 005054 0000 5080                DC.L    STR_2
0064
0065 005058 0007           STR_TBL3  DC.W    7               ;for STR_3
0066 00505A 0000 5086                DC.L    STR_3
0067
0068 00505E 0007           STR_TBL4  DC.W    7               ;for STR_4
0069 005060 0000 508D                DC.L    STR_4
0070
0071 005064 0006           STR_TBL5  DC.W    6               ;for STR_5
0072 005066 0000 5094                DC.L    STR_5
0073
0074 00506A 0005           STR_TBL6  DC.W    5               ;for STR_6
0075 00506C 0000 509A                DC.L    STR_6
0076
0077 005070 0004           STR_TBL7  DC.W    4               ;for STR_7
0078 005072 0000 509F                DC.L    STR_7
0079
0080                       *         --------------
0081                       *         test data area
0082                       *         --------------
0083 005076 4142 4344 45   STR_0     DC.B    "ABCDE"         ;STR_1 = STR_0   EQ(Z=1)
0084 00507B 4142 4344 45   STR_1     DC.B    "ABCDE"
0085
0086 005080 4142 4344 4546 STR_2     DC.B    "ABCDEF"        ;STR_3 > STR_2   HI(Z=0,C=0)
0087 005086 4142 4344 4546 STR_3     DC.B    "ABCDEFA"
            41
0088
0089 00508D 4142 4344 4546 STR_4     DC.B    "ABCDEFA"       ;STR_5 < STR_4   CS(C=1)
            41
0090 005094 4142 4344 4546 STR_5     DC.B    "ABCDEF"
0091
0092 00509A 5657 5859 5A   STR_6     DC.B    "VWXYZ"         ;STR_6 < STR_5   CS(C=1)
0093 00509F 5253 5455      STR_7     DC.B    "RSTU"
0094
0095        =00005000                END     S_CMP
```

第2部 プログラム

16 FIFO(Queue)

FIFO (First In First Out) は，データを一時的にプールしておくエリア（バッファ）へ順にデータを詰め込み，詰め込んだ順に取り出すような操作を意味しますが，Queue(キュー，待ち行列）とも呼ばれます．

ここではメモリ上にFIFOを実現しますが，プリンタ・スプーラからマルチ・タスクOSの分野まで広範囲に応用できます．

■ハードウェアとFIFO

画像処理などの分野では，大量のデータを高速に外部から読み込む必要があり，送り込まれるデータ量に対処できるだけの能力がコンピュータ側にない場合には，データを読み落とす可能性があります．そこで，外部からのデータをまずFIFOへ転送しFIFOからのデータを取り込めば，FIFOがフルになるまでの時間だけコンピュータ側に時間的余裕が与えられます．このように，処理中に外部から転送されたデータはFIFOへ格納されるので，コンピュータ自身の時間的遅れをFIFOに吸収できるわけです．

■ソフトウェアで実現するFIFO

単にバッファ・エリアへデータを書き込みそれを読み出す操作と異なり，書き込みと読み出しは非同期で行われます．しかも特定の時間帯に注目すると，バッファへの書き込みだけが連続して起動されたり，逆にバッファからの読み出しが連続して起動されることもあります．

例1： キーボード・バッファ

タイプ・アヘッドのサポートされているコンソール入力系では，キーボードの入力は割り込みで処理され，たとえディスク・アクセス中にヒットされたキーであっても無視されません．つまりキーボードから入力された文字は一度キーボード・バッファへ格納され，その後適当な時期にバッファから順に取り出され使用されます．

例2： プリンタ・スプーラ

コンピュータ・システムから転送される印字データは一度プリンタ内のキューへプールされ，プールされたデータを印字するようなプログラムがライン・プリンタ内に存在します．このため，プリンタ内のキューがフルになるまで〝ジャンジャン〟印字データを受け入れることができるので，少なくともプリンタ内のキューがフルになるまでは，高速に印字データをプリンタへ送り込むことができます．

いうまでもないことですが，機械部の動作スピード（プリンタ・ヘッドの移動時間には機械部が介在する）と電子のスピード（コンピュータ内部のスピード）とは次元が異なるわけですから，キューを介してコンピュータ～プリンタ間の通信をすることは，極めて有効な手段であるわけです．

1 メモリ上にFIFO(Queue)を実現する

◉サンプルプログラム ［FIFO］

下位番地方向から上位番地へ向かってキューを延ばす例ですが，キューは循環バッファとして機能するようにしました．

■動作

後述のようにプログラムは3つの部分から構成されるが，各ルーチンは**表2.16**のような動作を想定している．

表2.16　プログラムの動作

サブルーチン名	動　　作
B__INIT	頻繁に実行させるようなものではなく，システムが起動されたときなどにキューを初期化するものである．ただし必要ならいつでもキューを初期化してよい．
PUT__BUF	キューへデータを書き込む．通常は割り込みで起動されるので裏処理としての意味あいが強い（書き込みが正常に行われた否かのステータスが返される）．
GET__BUF	キューからデータを読み出す．通常は割り込みで起動されるので裏処理としての意味あいが強い（読み出しが正常に行われたか否かのステータスが返される）．

キューを実現するには2つのポインタと2つのステータスを管理しなければならない．

<PUT__PTR>　キューへデータを書き込むことを〝プット〟というが，書き込むべき場所を管理する場所がPUT__PTRであり，ここにポインタを格納しておく．

D0の内容(サイズをバイトとする)をキューへ書き込むには，以下のようにする．

```
MOVEA.L   PUT__PTR, A0 : PUT__PTRからポインタをA0へ取り出す
MOVE.B    D0, (A0)     : キューへ書き込む
```

<GET__PTR>　キューからデータを読み込むことを〝ゲット〟というが，読み込むべき場所を管理する場所がGET__PTRであり，ここにポインタを格納しておく．キューからD0（サイズをバイトとする）へデータを読み込むには，以下のようにすればよい．

```
MOVEA.L   GET__PTR, A0 : GET__PTRからポインタをA0へ取り出す
MOVE.B    (A0), D0     : キューからD0へ読み込む
```

<PUT__FLG>　ここにはキューへ書き込み動作を管理するステータスを格納しておく．PUT__BUFが起動された際に書き込む余地がないかも知れないので，書き込み許可／禁止ステータスの管理が必要である.PUT__FLGの意味を次のように定義している．

表2.17 PUT__FLGの意味

ステータス	意　　味
$00	書き込み禁止（buffer full）
$FF	書き込み許可

〈GET_FLG〉 ここにはキューからの読み込み動作を管理するステータスを格納しておく．GET_BUFが起動された際に読み出すべき内容があるとは限らないので，読み出し許可／禁止ステータスの管理が必要である．GET_FLGの意味を次のように定義している．

表2.18
GET_FLGの意味

ステータス	意　　　味
＄00	読み出し禁止（buffer empty）
＄FF	読み出し許可

キューのロケーションは次のような記号番地（ラベル）で表現するが，データをキューの最後まで書き込むとポインタを先頭へリンクするので，低位アドレスとか高位アドレスという概念はない．

BF_START　：キューの先頭（低位アドレス）
BF_END　：キューの最後（高位アドレス）

以下，ポインタやステータスをどのように操作すればよいか，という本題に入る．

■初期化〈B_INIT〉

キュー（バッファ）の初期化とはポインタとステータスに初期値を与えることである．

1．ポインタの初期化

① PUT_PTRへキューの先頭アドレス（BF_START）を格納し，最初にPUT_BUFが呼び出されたときにキューの先頭からデータを格納できるようにする．

② GET_PTRへキューの先頭アドレス（BF_START）を格納し，最初GET_BUFが呼び出されたときにキューの先頭からデータを取り出せるようにする．

2．ステータスの初期化

① PUT_FLGへ＄FFを格納し，PUT_BUFが呼び出されたときにキューへの書き込みができるようにする．

② GET_FLGへ＄00を格納し、GET_BUFがいきなり呼び出されても無意味な読み込みをしないようにする．

■キューへの書き込み〈PUT_BUF〉

1．PUT_FLGのチェック

キューへの書き込みステータスをPUT_FLGでチェックし，書き込み禁止なら何もしない．このステータスはメインへの情報として返されるので，呼び出し側ではキューへの書き込みが行われたのか拒否されたのかを知ることができる．

2．PUT_PTRでポイントされるロケーションへデータを書き込む

書き込み可ならPUT_PTRからポインタを取り出し，その場所へメインから渡されたデータを書き込む．

キューは有限長なので次のような操作が要求される．

3．PUT_PTRを更新

すでにデータをキューへ書き込んだのでPUT_PTRを更新し，次の書き込みに備えなければならない．そこで，今書き込んだロケーションがキューの最後（BF_END）であるかどうかをチェックし，先がなければPUT_PTRをキューの先頭（BF_START）へリンクし，そうでなければ単にPUT_PTRを先へ進める．

4．PUT_FLG，GET_FLGの設定

もしGET_BUFがまったく起動されず書き込みだけが起動された場合，取り出されるべ

き意味のあるデータがキューに入っているはずであるから，次にPUT__BUFが起動されると待ち行列の内容が破壊されるかもしれない．そこで，PUT__PTRとGET__PTRとを比較し，一致していれば書き込み禁止とする．

いずれにしてもデータを書き込んだのだから，次に起動されるGET__BUFのために読み出しを許可する．

■キューからの読み込み〈GET_BUF〉

基本的な考え方はすべてPUT__BUFと同様であり，読み込みが許可されているかをチェックし，読み込みに成功したらポインタの更新やステータスの設定など，次回に呼び出された場合に対処するための準備をする．

1. GET__FLGのチェック

GET__FLGの内容そのものは呼び出し側へ返されるが,読み込みが許可されていなければ何もしない．

2. GET__PTRでポイントされるロケーションからデータを読み込む

3. GET__PTRを更新

今読み込んだロケーションがBF__ENDならGET__PTRをキュー先頭（BF__START）へリンクし，そうでなければ単にGET__PTRを先に進める．

4. PUT__FLG，GET__FLGの設定

読み出し操作だけが連続して起動された場合，やがて意味のないロケーションからの読み出しをしてしまう可能性がある．そこで，GET__PTRとPUT__PTRを比較し，一致していれば以後の読み出しを禁止する．いずれにしても読み出しを実行したのだから書き込みは許可する．

■各行の意味

行16　：キューの初期化

行22〜24：Ｄ０の下位バイトへプットすべきデータを格納し，スタックへワード・サイズでプッシュしPUT__BUFを呼び出す．その後にＤ１にステータスを受け取っている．

行30〜33：空の内容（ロング・ワード）をスタックへプッシュしGET__BUFを呼び出す．その後Ｄ０にデータ、Ｄ１にステータスを受け取っている．

以後の行についてはリストとこれまでの説明で十分であると判断し，解説は割愛する．

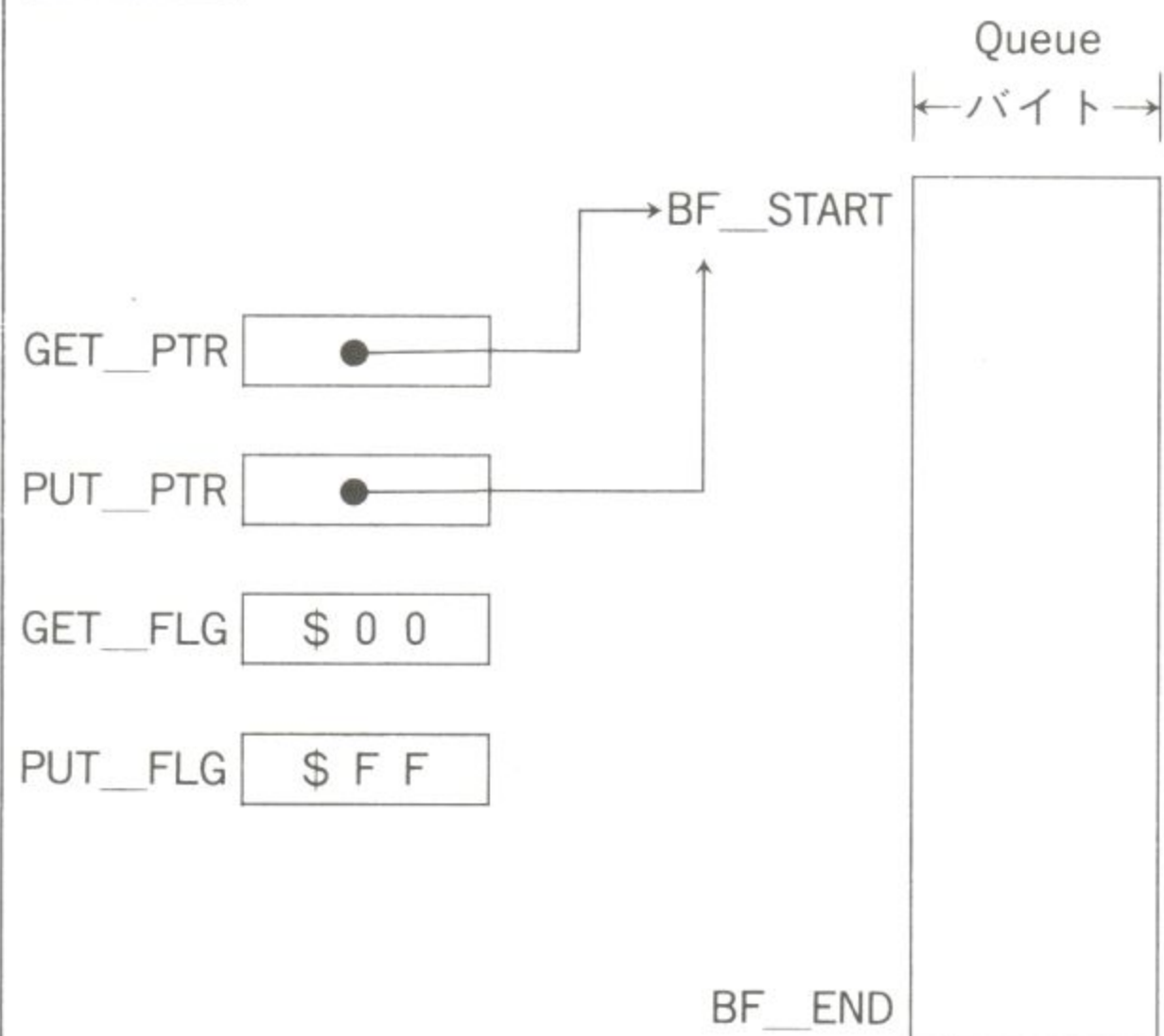
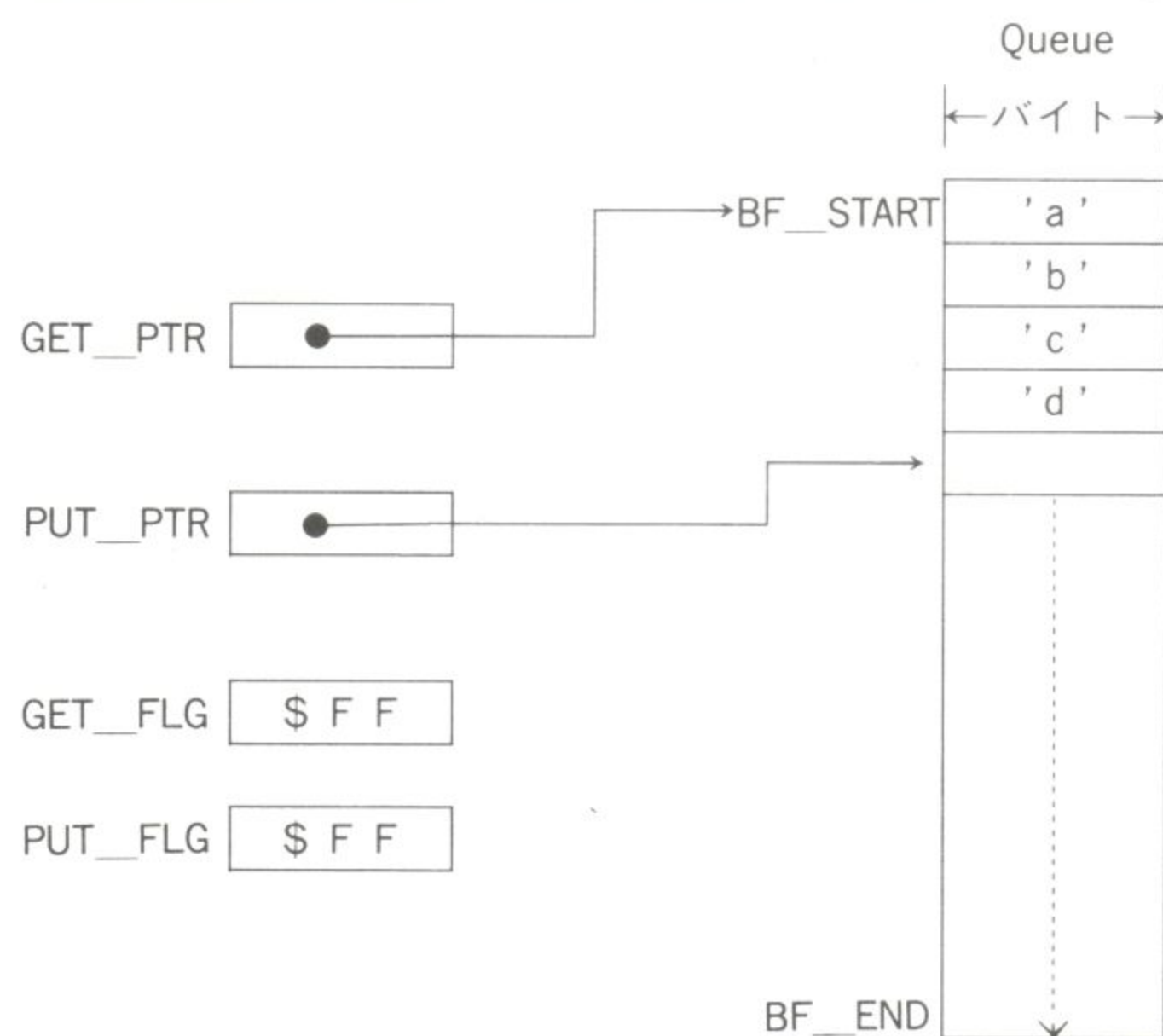
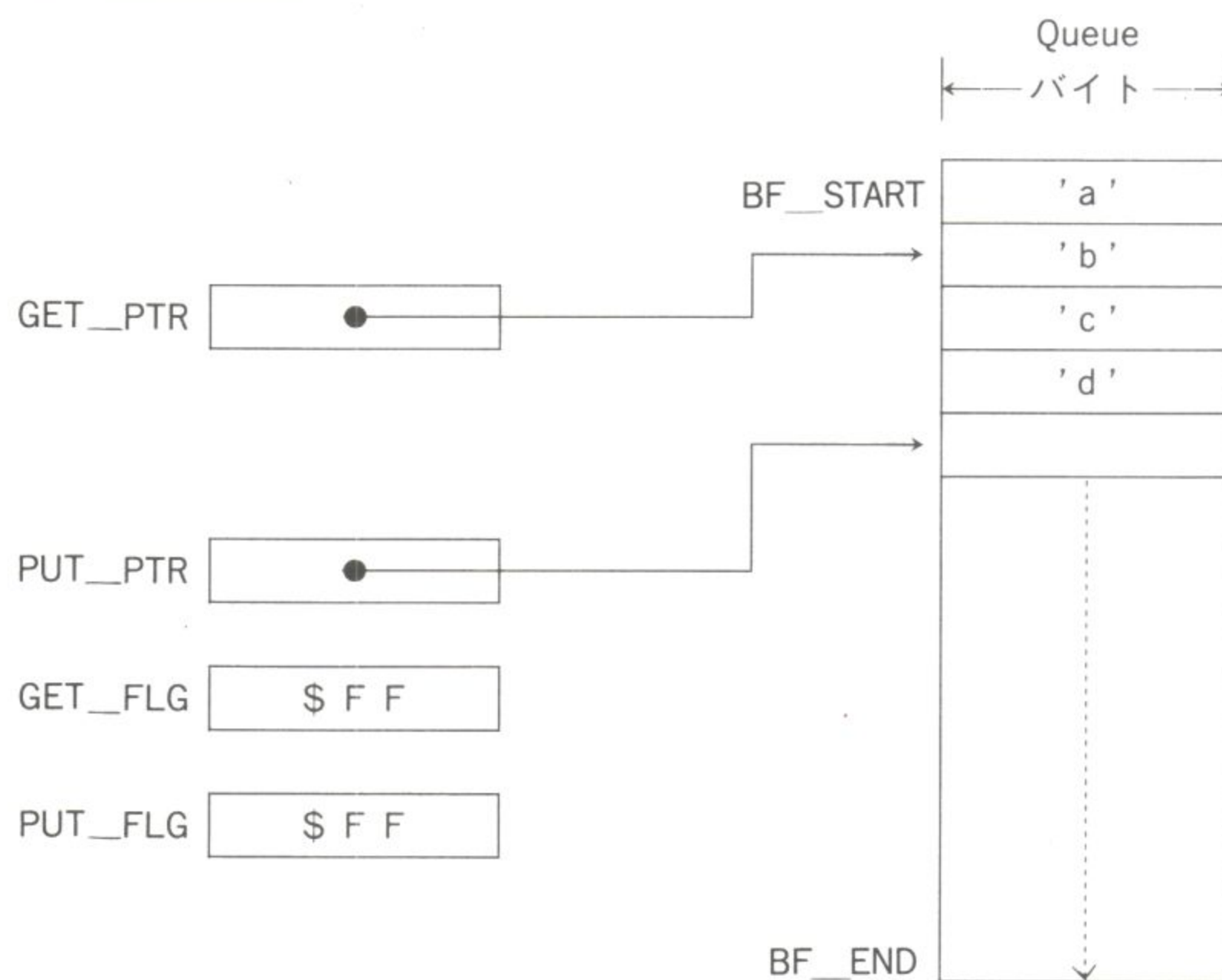
アプリケーション・ヒント

本例ではキューの長さは４バイトでそのデータ・サイズはバイトである．

これは確認するのに適当であると判断したからであるが，プログラムは割り込みに依存する部分を除きフルセットである．またキューのサイズや先頭／終了アドレスをダイナミックに管理するには，そのための変数領域を確保し，必要に応じてそれらを取り出せばよい．

B__INIT，PUT__BUF，GET__BUFの各ルーチンは,ポインタやステータス・フラグを適切に管理するために割り込み禁止状態で実行すべきである(B__INITが実行されるまではPUT__BUFやGET__BUFが起動されてはならない).したがってこれらはOSやシステム・モニタなどでサポートされるべきもので，ユーザ状態で走るプログラムとはちょっと異なる．

プリンタ・スプーラだけを意識した機器組み込み用に68000を応用するのであれば，豊富なアドレス・レジスタにポインタ，データ・レジスタにデータ自身やステータス・フラグを割り当てれば，インフォメーション領域は不要である．またこれらの領域をローカル・エリア上に確保できるなど，他のプロセッサの基本命令では実現できないことも可能である．

図2.46　FIFOの様子

note：①PUT__PTRという変数エリアは，キューへの書き込み要求が発生したときに，「どこへ書き込むべきか」というアドレスを格納するもの．

②GET__PTRという変数エリアは，キューからの読み込み要求が発生したときに，「どこから読み込むべきか」というアドレスを格納するもの．

③キューはBF__STARTからBF__ENDへ向って成長するので，BF__ENDまで来たらBF__STARTへリンクしてやる．これは，GET__PTRやPUT__PTRの内容をそのように書き換えることを意味する．

④キューへのputばかり起動されると，やがてgetすべき位置まで来てしまう．このような場合には，次回のputをウエイトさせるようにステータスを操作すればよい．

⑤キューへのgetばかり起動されると，やがてputすべき位置まで来てしまう．このような場合には，次回のgetをウエイトさせるようにステータスを操作すればよい．

図2.47　FIFOのフローチャート〈1〉

初期化：B__INIT

キューの先頭アドレス→(PUT__PTR)
キューの先頭アドレス→(GET__PTR)
$FF　→(PUT__FLG)
$00　→(GET__FLG)

終　了

キューへの書き込み：PUT__BUF

(PUT__FLG)＝$00　Yes / NO

(PUT__PTR)　→A0
メインからの引数→D0

キューへの書き込み：D0→(A0)
アドレスを進める　：A0＝A0＋1

A0＞BF__END　NO / YES

次に put すべきアドレスの決定に必要

Put Pointer を先頭へリンク：
BF__START → A0

更新したポインタを格納：
A0 → (PUT__PTR)

A0＝(GET__PTR)　NO / YES

次回の書き込みを禁止：
$00 →(PUT__FLG)

次回の読み出しを許可：
$FF →(GET__FLG)

PUT__FLG の内容をメインへ返す

終　了

●PUT__BUFとスタック

ワード

＋10　＋11　data

①PUT__BUF はdataをスタックへプッシュしてから呼び出される．
②PUT__BUF からリターンする時には，dataの位置にステータス･フラグがもどされる．
③スタック上のディスプレースメントは，LINK 後の A6 で与えられる．

図2.48　FIFOのフローチャート〈2〉

キューからの読み込み：GET_BUF

(GET_FLG)＝$00

YES

NO

(GET_PTR) → A0

●キューからデータを読み込んで引数エリアへ書き込む．
●アドレスを進める：A0＝A0＋1

A0＞BF_END

NO

次に get すべきアドレスの決定に必要

YES

get pointer を先頭へリンク：
BF_START → A0

更新したポインタを格納：
A0 → (GET_PTR)

A0＝(PUT_PTR)

NO

YES

次回の読み込みを禁止：
$00 → (GET_FLG)

次回の書き込みを許可：
$FF → (PUT_FLG)

GET_FLGの内容をメインへ返す

終　了

●GET_BUFとスタック

ワード

＋10	＋11 data
＋12	＋13 status

①メインからは何もわたされない．
②GET_BUF からリターンするときには，data の位置に読み込んだ内容，status の位置にステータス・フラグがもどされる．
③スタック上のディスプレースメントは，LINK 後の A6 で与えられる．

リスト[FIFO]

```
LINE ADDR. CODE/VALUE          LABEL    OP       OPERAND         COMMENT

0005
0006        =00000004          B_SIZE   EQU      4
0007        =000000FF          FLG_ON   EQU      $FF
0008
0010        =00005000                   ORG      $5000
0011        =00005000          FIFO
0012                           *                 ***********************
0013                           *                 test initialize routine
0014                           *                 ***********************
0015
0016 005000 4EB9 0000 501C              JSR      B_INIT
0017 005006                    BREAK_0
0018                           *                 **********************************
0019                           *                 test put character to buffer area
0020                           *                 **********************************
0021
0022 005006 3F00                        MOVE.W   D0,-(SP)        ;put D0.B to buffer
0023 005008 4EB9 0000 5042              JSR      PUT_BUF
0024 00500E 321F                        MOVE.W   (SP)+,D1        ;return status flag to D1
0025 005010                    BREAK_1
0026                           *                 ***********************************
0027                           *                 test get character from buffer area
0028                           *                 ***********************************
0029
0030 005010 2F00                        MOVE.L   D0,-(SP)        ;dummy
0031 005012 4EB9 0000 5096              JSR      GET_BUF
0032 005018 301F                        MOVE.W   (SP)+,D0        ;get data
0033 00501A 321F                        MOVE.W   (SP)+,D1        ;get status
0034 00501C                    BREAK_2
0035                           *                 ----------------
0036                           *                 [sub] Initialize
0037                           *                 ----------------
0038 00501C                    B_INIT
0039 00501C 40E7                        MOVE.W   SR,-(SP)
0040
0041 00501E 23FC 0000 50F2              MOVE.L   #BF_START,PUT_PTR ;ini. put pointer
            0000 50E8
0042 005028 23FC 0000 50F2              MOVE.L   #BF_START,GET_PTR ;ini. get pointer
            0000 50EC
0043 005032 13FC 00FF 0000              MOVE.B   #FLG_ON,PUT_FLG ;set put flag
            50F0
0044 00503A 4239 0000 50F1              CLR.B    GET_FLG         ;clear get flag(buffer empty)
0045
0046 005040 4E77                        RTR
0047
0048                           *                 ---------------------------------
0049                           *                 [sub] put character to buffer area
0050                           *                 ---------------------------------
0051 005042                    PUT_BUF
0052 005042 40E7                        MOVE.W   SR,-(SP)
0053 005044 4E56 0000                   LINK     A6,#0
0054 005048 48E7 C080                   MOVEM.L  D0-D1/A0,-(SP)
0055
0056                           *                 /* check put status */
0057
0058 00504C 1239 0000 50F0              MOVE.B   PUT_FLG,D1      ;test put flag
0059 005052 6736                        BEQ      ESC_BPUT
0060
0061                           *                 /* put data to queue */
0062
0063 005054 2079 0000 50E8              MOVEA.L  PUT_PTR,A0      ;A0:put pointer
0064 00505A 302E 000A                   MOVE.W   10(A6),D0       ;load argument to D0
0065 00505E 10C0                        MOVE.B   D0,(A0)+        ;put data to buffer
0066
0067                           *                 /* next put address */
0068
0069 005060 B1FC 0000 50F5              CMPA.L   #BF_END,A0      ;   (A0,put pointer) - (BF_END)
0070 005066 6306                        BLS      CHK_ADR0        ;LS:(A0,put pointer) <= (BF_END)
0071 005068 41F9 0000 50F2              LEA      BF_START,A0     ;A0:buffer start address
0072
0073                           *                 /* set/rerest status flag */
0074 00506E                    CHK_ADR0
0075 00506E B1F9 0000 50EC              CMPA.L   GET_PTR,A0      ;put pointer == get pointer
0076 005074 6606                        BNE      GET_RDY
0077 005076 4239 0000 50F0              CLR.B    PUT_FLG         ;buffer full(can't put)
0078 00507C                    GET_RDY
0079 00507C 13FC 00FF 0000              MOVE.B   #FLG_ON,GET_FLG
            50F1
```

第2部 プログラム

```
0080 005084 23C8 0000 50E8            MOVE.L  A0,PUT_PTR      ;re-load put pointer to PUT_PTR
0081
0082                        *                 /* return job */
0083 00508A                 ESC_BPUT
0084 00508A 3D41 000A                 MOVE.W  D1,10(A6)       ;return status flag to main
0085
0086 00508E 4CDF 0103                 MOVEM.L (SP)+,D0-D1/A0
0087 005092 4E5E                      UNLK    A6
0088 005094 4E77                      RTR
0089
0090                        *         -----------------------------------
0091                        *         [sub] get character form buffer area
0092                        *         -----------------------------------
0093 005096                 GET_BUF
0094 005096 40E7                      MOVE.W  SR,-(SP)
0095 005098 4E56 0000                 LINK    A6,#0
0096 00509C 48E7 4080                 MOVEM.L D1/A0,-(SP)
0097
0098                        *                 /* check get status */
0099
0100 0050A0 1239 0000 50F1            MOVE.B  GET_FLG,D1      ;test get flag
0101 0050A6 6734                      BEQ     ESC_BGET
0102
0103                        *                 /* get data from queue */
0104
0105 0050A8 2079 0000 50EC            MOVEA.L GET_PTR,A0      ;A0:get pointer
0106 0050AE 1D58 000B                 MOVE.B  (A0)+,11(A6)    ;return data to main
0107
0108                        *                 /* next get address */
0109
0110 0050B2 B1FC 0000 50F5            CMPA.L  #BF_END,A0      ;   (A0,get pointer) - (BF_END)
0111 0050B8 6306                      BLS     CHK_ADR1        ;LS:(A0,get pointer) <= (BF_END)
0112 0050BA 41F9 0000 50F2            LEA     BF_START,A0     ;A0:buffer start address
0113
0114                        *                 /* set/reset status flag */
0115 0050C0                 CHK_ADR1
0116 0050C0 B1F9 0000 50E8            CMPA.L  PUT_PTR,A0
0117 0050C6 6606                      BNE     PUT_RDY
0118 0050C8 4239 0000 50F1            CLR.B   GET_FLG         ;buffer empty(can't get)
0119 0050CE                 PUT_RDY
0120 0050CE 13FC 00FF 0000            MOVE.B  #FLG_ON,PUT_FLG
            50F0
0121 0050D6 23C8 0000 50EC            MOVE.L  A0,GET_PTR
0122
0123                        *                 /* returun job */
0124 0050DC                 ESC_BGET
0125 0050DC 1D41 000D                 MOVE.B  D1,13(A6)       ;return status to main
0126
0127 0050E0 4CDF 0102                 MOVEM.L (SP)+,D1/A0
0128 0050E4 4E5E                      UNLK    A6
0129 0050E6 4E77                      RTR
0130
0131                        *         ----------------
0132                        *         information area
0133                        *         ----------------
0134
0135 0050E8 =00000004       PUT_PTR   DS.L    1               ;put pointer of buffer area
0136 0050EC =00000004       GET_PTR   DS.L    1               ;get pointer of buffer area
0137 0050F0 =00000004
0138 0050F0 =00000001       PUT_FLG   DS.B    1               ;(0)buffer full   (1)put ready
0139 0050F1 =00000001       GET_FLG   DS.B    1               ;(0)buffer empty  (1)get ready
0140
0141                        *         ------------------------
0142                        *         FIFO (Queue) Buffer Area
0143                        *         ------------------------
0144
0145 0050F2 =00000003       BF_START DS.B     B_SIZE-1
0146 0050F5 =00000001       BF_END   DS.B     1
0147
0148        =00005000                END      FIFO
```

17 コード変換

コード変換は，「コンピュータ内部に要求される表現」と「文字」とのインターフェースを行うもので，我々の扱う数値表現が文字そのものであることによります．

例 1： 16進 ──→ バイナリ

キーボードからアドレスを取り込んでそれをコンピュータ内部の演算に使用する場合，オペレータが〝FF〞という2文字を入力した場合，実際にはアスキー・コードの`F´を意味する＄46として入力されます．これはキーボード上の〝エフ〞がヒットされると，＄46というコードに置き換えられるからです．

そこで〝FF〞2文字を1バイトの内部表現（バイナリ値）に変換してから，目的の処理をしなければなりません．

例 2： バイナリ ──→ 16進

メモリの内部をスクリーンやプリンタへ出力する場合などに必要です．

今メモリ内に＄FFという値（バイナリ値）が格納されているとき，〝エフ〞という文字コード2文字に変換してスクリーンやプリンタへ出力しなければなりません．

以上のように我々が扱えるものは〝文字〞であり，コンピュータの理解できるものは数値であることに注意しなければなりません．

サブルーチンへの引数の渡し方，結果の受け取り方，サブルーチン自身の処理などは，現場での要求に応じて少々の変更が必要かもしれません．そのためリストを熟読され，「何をしているのか」ということを理解してほしいと思います．

1 ビット列を意味する文字列を内部表現(バイナリ)に変換する

◉サンプルプログラム ［ABIN__BIN］

アセンブラでの記述にはビット列で記述した方がよいこともあります．
たとえば，

```
VOL     EQU     %10101010
```

なる行はVOLという定数に＄AAという値が与えられますが，ソース自身は文字列表現なので，言語の処理系にはここで取り上げるような変換ルーチンが存在します．

■動作

`1´と`0´から構成される文字列の先頭アドレスとその長さをスタックへプッシュし，ABIN_BINを呼び出す．結果は常にロング・ワードで返されるので，長さが32に達しなかった場合でも上位に＄0を満たすように処理される．

■解法

まず32ビットの作業用レジスタをクリアしておき，"0"ならそのまま，"1"ならビットを立てる．この操作を左シフトしながら文字数だけ繰り返せばよい．

■各行の意味

行10〜12：メッセージの出力を行うもので，A0にメッセージが格納されている先頭アドレスをセットする（本モニタの機能9）．

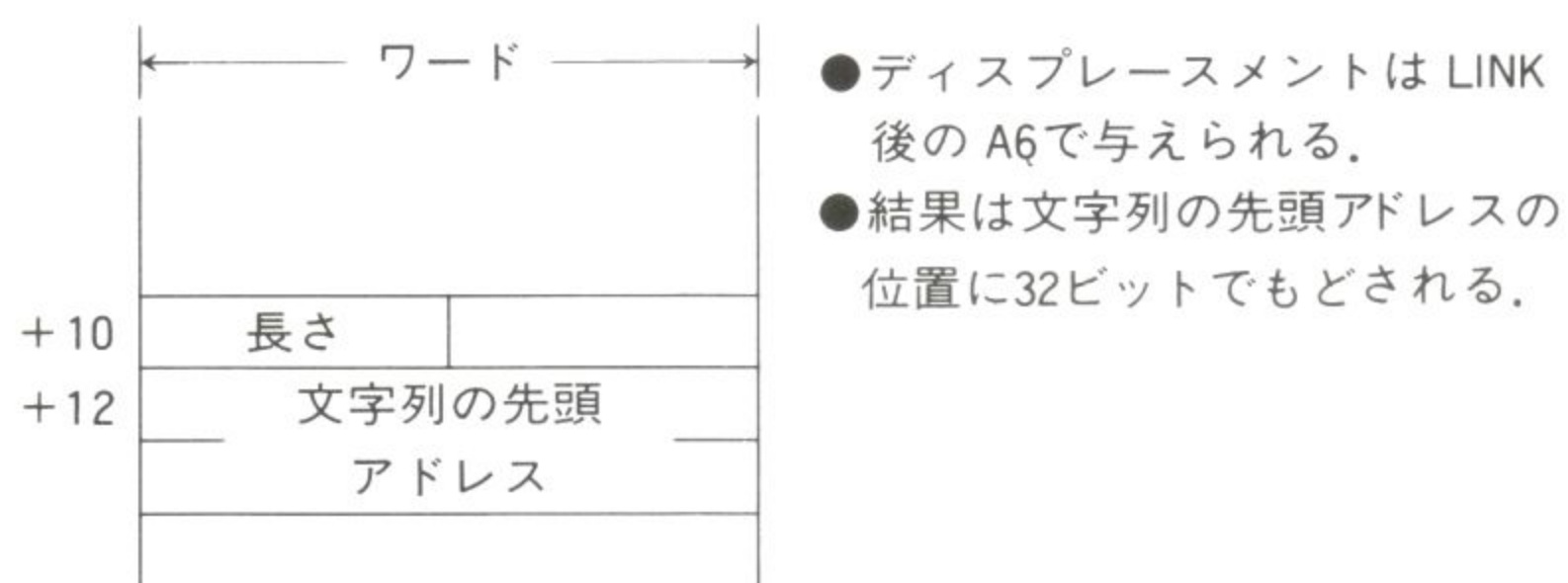
行14〜17：1行の入力を行うもので，D1に取り込む文字数＋1をセットする（本モニタの機能＄A）．

行23〜26：引数として文字列の先頭アドレス，文字列の長さ，の順でスタックへプッシュしABIN_BINを呼び出す．24行ではバイト・サイズでシステム・スタックへプッシュしているが，SPは2つだけ減じられる．

行40〜61：サブルーチン部である．メインから渡される文字列長はバイト・サイズなので，まずレジスタをクリアしてから受け取っている．

アプリケーション・ヒント

実際には"1"や"0"が入力されるとは限らないので，不当文字の入力がなかったかどうかをチェックするなど，それなりのエラー処理も要求される．

図2.49 ABIN__BINとスタック

リスト［ABIN_BIN］

```
LINE ADDR.  CODE/VALUE          LABEL    OP        OPERAND          COMMENT

0004                                     NAM       ABIN_BIN.A68
0005
0006
0007         =00005000                   ORG       $5000
0008
0009         =00005000          ENTRY
0010 005000 103C 0009                    MOVE.B    #9,D0            ;display strings
0011 005004 41F9 0000 50A5               LEA       MSG,A0
0012 00500A 4E40                         TRAP      #0
0013
0014 00500C 123C 0021                    MOVE.B    #32+1,D1         ;length + cr
0015 005010 103C 000A                    MOVE.B    #$A,D0           ;line input
0016 005014 41F9 0000 5082               LEA.L     LIN_BUF,A0
0017 00501A 4E40                         TRAP      #0
0018
0019 00501C 6148                         BSR       CRLF_OUT
0020
0021                            *                  /* convert */
0022
0023 00501E 4879 0000 5084               PEA       STR_BUF          ;push string address
0024 005024 1F28 0001                    MOVE.B    1(A0),-(SP)      ;push string length
0025 005028                     BREAK0
0026 005028 610A                         BSR       ABIN_BIN         ;call
0027 00502A                     BREAK1
0028 00502A 321F                         MOVE.W    (SP)+,D1         ;result(dummy)
0029 00502C 221F                         MOVE.L    (SP)+,D1         ;result
0030 00502E                     BREAK2
0031 00502E 103C 0000                    MOVE.B    #0,D0            ;return to system
0032 005032 4E40                         TRAP      #0
0033
0034                            *                  D0 : loop counter
0035                            *                  D1 : work register for converting
0036                            *                  D2 : work register for reading
0037                            *                  A0 : string pointer
0038
0039 005034                     ABIN_BIN
0040 005034 40E7                         MOVE.W    SR,-(SP)         ;link
0041 005036 4E56 0000                    LINK      A6,#0
0042 00503A 48E7 E080                    MOVEM.L   D0-D2/A0,-(SP)
0043
0044 00503E 4280                         CLR.L     D0               ;initialize
0045 005040 4281                         CLR.L     D1
0046
0047 005042 102E 000A                    MOVE.B    10(A6),D0        ;setup loop counter
0048 005046 206E 000C                    MOVEA.L   12(A6),A0        ;setup address pointer
0049 00504A 5380                         SUBQ.L    #1,D0            ;adjust loop counter
0050 00504C                     ABIN_LP
0051 00504C E381                         ASL.L     #1,D1            ;arithmetic shift left
0052 00504E 1418                         MOVE.B    (A0)+,D2         ;read string
0053 005050 0402 0030                    SUBI.B    #'0',D2          ;D2.B = 0 or 1
0054 005054 8202                         OR.B      D2,D1
0055 005056 51C8 FFF4                    DBRA      D0,ABIN_LP
0056
0057 00505A 2D41 000C                    MOVE.L    D1,12(A6)        ;retrun arg.
0058
0059 00505E 4CDF 0107                    MOVEM.L   (SP)+,D0-D2/A0   ;unlk
0060 005062 4E5E                         UNLK      A6
0061 005064 4E77                         RTR                        ;return
0062
0063                            *
0064                            * CR/LF
0065                            *
0066 005066                     CRLF_OUT
0067 005066 40E7                         MOVE.W    SR,-(SP)
0068 005068 48E7 C000                    MOVEM.L   D0-D1,-(SP)
0069
0070 00506C 103C 0002                    MOVE.B    #2,D0
0071 005070 123C 000D                    MOVE.B    #$0D,D1
0072 005074 4E40                         TRAP      #0
0073 005076 123C 000A                    MOVE.B    #$0A,D1
0074 00507A 4E40                         TRAP      #0
0075
0076 00507C 4CDF 0003                    MOVEM.L   (SP)+,D0-D1
0077 005080 4E77                         RTR
0078
0079                            *-------------
0080                            * string area
0081                            *-------------
```

```
0082                           *             /* for funcion $0A */
0083
0084                                  EVEN
0085 005082 21                LIN_BUF  DC.B    32+1            ;buffer length (len + cr)
0086 005083 =00000001                   DS.B    1               ;for system
0087 005084 =00000021          STR_BUF  DS.B    32+1            ;string area
0088
0089                           *             /* message */
0090
0091 0050A5 4269 7420 6461     MSG      DC.B    "Bit data ? _",'$'
            7461 203F 205F
            24
0092 0050B2
0093        =00005000                   END     ENTRY
```

2 16進文字列をバイナリ表現に変換する

◉サンプルプログラム ［AHEX__BIN］

■動作

\`0´～\`9´，\`A´～\`F´の文字セットから構成される文字列の先頭アドレスとその長さをスタックへプッシュし，AHEX__BINを呼び出す．結果は常にロング・ワードで返されるので，長さが8文字に達しなかった場合でも上位に＄0を満たすように処理される．

■解法

16進1文字は1バイトの記憶空間を必要とし，バイナリ表現では4ビットに変換される．そこで4ビット（これをニブルという）に変換する過程が先決となる．

まずアスキー・コードに注目してみる．

表2.19
文字・文字コード・変数値

文　字	文字コード	変換値（内部表現）
\`0´～\`9´	＄30～＄39	＄0～＄9
\`A´～\`F´	＄41～＄46	＄A～＄F

よって1文字読み込んだら，まず＄30を減じる．この値が＄Aより小さければ元の文字は\`0´～\`9´のいずれかであり変換は完了する．そうでなければさらに＄7を減じる．

以上の操作を4ビット左へシフトしながら文字数だけ繰り返せばよい．

■各行の意味

行10～19：本モニタの機能を利用して16進文字列の取り込みを行う．

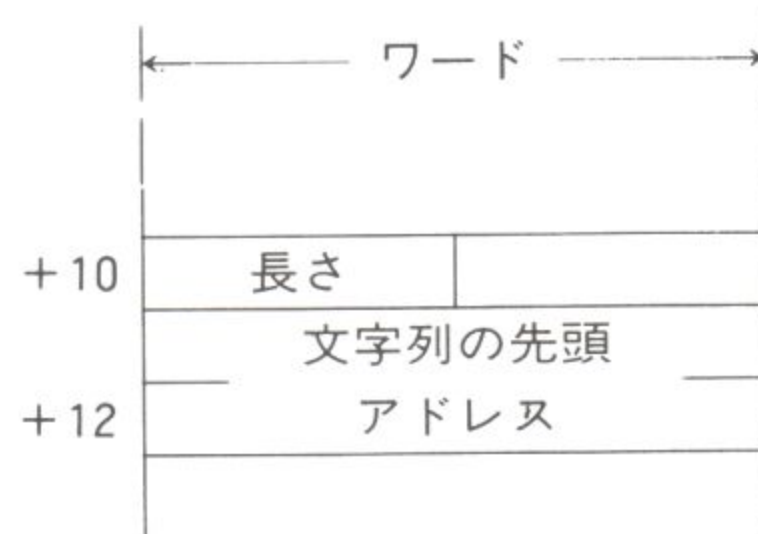
行23～26：引数として16進文字列の先頭アドレス，文字列の長さをスタックへプッシュしAHEX__BINを呼び出す．

行28～29：変換値をD1（ロング・ワード）で受けスタックを復元している．

行46～65：変換ルーチンであるが，16進1文字を4ビットに変換するルーチンCONV__BIN（68行～76行）を呼び出している．

アプリケーション・ヒント

実際には16進文字だけが入力されるとは限らないし，\`a´～\`f´までの文字は大文字に変換してから本ルーチンを呼び出すべきである．このようにしないと実用的な処理とはいえない．

図2.50　AHEX__BINとスタック

リスト[AHEX_BIN]

```
LINE ADDR. CODE/VALUE         LABEL     OP       OPERAND          COMMENT

0004                                    NAM      AHEX_BIN.A68
0005
0006
0007        =00005000                   ORG      $5000
0008
0009        =00005000         ENTRY
0010 005000 103C 0009                   MOVE.B   #9,D0            ;display message
0011 005004 41F9 0000 509D              LEA      MSG,A0
0012 00500A 4E40                        TRAP     #0
0013
0014 00500C 123C 0009                   MOVE.B   #8+1,D1          ;get hex strings
0015 005010 103C 000A                   MOVE.B   #$A,D0
0016 005014 41F9 0000 5092              LEA      LIN_BUF,A0
0017 00501A 4E40                        TRAP     #0
0018
0019 00501C 6158                        BSR      CRLF_OUT
0020
0021                          *                 /* convert */
0022
0023 00501E 4879 0000 5094              PEA      STR_BUF          ;push string address
0024 005024 1F28 0001                   MOVE.B   1(A0),-(SP)      ;push string length
0025 005028                   BREAK0
0026 005028 610C                        BSR      AHEX_BIN
0027 00502A                   BREAK1
0028 00502A 222F 0002                   MOVE.L   2(SP),D1         ;receive data
0029 00502E 5C8F                        ADDQ.L   #6,SP            ;adjust stack pointer
0030 005030                   BREAK2
0031 005030 103C 0000                   MOVE.B   #0,D0            ;return to system
0032 005034 4E40                        TRAP     #0
0033
0034
0035                          *                 ------------------------------------------
0036                          *                  convert ascii hex string into binary
0037                          *
0038                          *                         D0 : loop counter
0039                          *                         D1 : work register for converting
0040                          *                         D2 : work register for reading
0041                          *                         A0 : string pointer
0042                          *                 ------------------------------------------
0043
0044                                    EVEN
0045                          AHEX_BIN
0046 005036 40E7                        MOVE.W   SR,-(SP)
0047 005038 4E56 0000                   LINK     A6,#0
0048 00503C 48E7 E080                   MOVEM.L  D0-D2/A0,-(SP)
0049
0050 005040 4280                        CLR.L    D0
0051 005042 4281                        CLR.L    D1
0052
0053 005044 102E 000A                   MOVE.B   10(A6),D0        ;set loop counter
0054 005048 206E 000C                   MOVEA.L  12(A6),A0        ;set address pointer
0055 00504C 5340                        SUBQ.W   #1,D0            ;adjust loop counter
0056 00504E                   XNBL_LP
0057 00504E 1418                        MOVE.B   (A0)+,D2
0058 005050 6110                        BSR      CONV_BIN
0059 005052 51C8 FFFA                   DBRA     D0,XNBL_LP
0060
0061 005056 2D41 000C                   MOVE.L   D1,12(A6)        ;return arg.
0062
0063 00505A 4CDF 0107                   MOVEM.L  (SP)+,D0-D2/A0
0064 00505E 4E5E                        UNLK     A6
0065 005060 4E77                        RTR
0066 005062
0067                          *                 /* convert into nibble */
0068 005062                   CONV_BIN
0069 005062 E989                        LSL.L    #4,D1            ;logical shift left for 4 bit
0070 005064 0402 0030                   SUBI.B   #'0',D2          ;convert 0 to 15($F)
0071 005068 0C02 000A                   CMPI.B   #10,D2           ;test D2 < 10
0072 00506C 6B04                        BMI      WHEN_09          ;Minus means already 0 to 9
0073 00506E 0402 0007                   SUBI.B   #7,D2            ;convert 10($A) to 15($F)
0074 005072                   WHEN_09
0075 005072 8202                        OR.B     D2,D1            ;get result to D1
0076 005074 4E75                        RTS
0077
0078                          *
0079                          * output CR/LF
0080                          *
0081 005076                   CRLF_OUT
```

```
0082 005076 40E7                       MOVE.W  SR,-(SP)        ;save registers
0083 005078 48E7 C000                  MOVEM.L D0-D1,-(SP)
0084
0085 00507C 103C 0002                  MOVE.B  #2,D0           ;cr/lf
0086 005080 123C 000D                  MOVE.B  #$0D,D1
0087 005084 4E40                       TRAP    #0
0088 005086 123C 000A                  MOVE.B  #$0A,D1
0089 00508A 4E40                       TRAP    #0
0090
0091 00508C 4CDF 0003                  MOVEM.L (SP)+,D0-D1     ;return registers
0092 005090 4E77                       RTR
0093
0094                          *
0095                          * string area
0096                          *
0097                                   EVEN
0098 005092 09                LIN_BUF  DC.B    8+1
0099 005093 =00000001                  DS.B    1
0100 005094 =00000009         STR_BUF  DS.B    8+1
0101
0102 00509D 4845 5820 7374    MSG      DC.B    "HEX string ? _",'$'
            7269 6E67 203F
            205F 24
0103
0104
0105        =00005000                  END     ENTRY
```

3 10進文字列をバイナリ表現に変換する

◉サンプルプログラム ［ADEC_BIN］

■動作

`0´～`9´の文字セットから構成される文字列の先頭アドレスとその長さをスタックへプッシュし，ADEC_BINを呼び出す．結果は常にロング・ワードで返されるので，長さが10文字に達しなかった場合でも上位に＄0を満たすように処理される．

■解法

10進文字列の持つ桁の重みは10であるので，10倍して加える操作を文字列の長さだけ実行し，その総数が求める値（内部表現）となる．

文字列の長さだけループすることになるが，10倍するという操作は桁移動の意味あいを含んでいる．

■各行の意味

行10～19：本モニタの機能を利用して10進文字列の取り込みをしている．

行23～29：引数として，10進文字列の先頭アドレス，文字列長をスタックへプッシュし，ADEC_BINを呼び出す．結果はＤ１にロング・ワードで取り出される．

行60～63：桁に10倍している部分で，２倍と８倍したものを加算して結果的に10倍している（重み付けをしている）．

行65～68：Ｄ２に取り出した`0´～`9´までの文字を＄０～＄９に変換し，桁の重みを加算する．この操作を文字列の長さだけ実行する．

アプリケーション・ヒント

実際には10進文字だけが入力されるとは限らないので，本ルーチンを呼び出す前にトラップすべきである．

リスト［ADEC_BIN］

```
LINE ADDR. CODE/VALUE        LABEL    OP       OPERAND         COMMENT

0004                                  NAM      ADEC_BIN.A68
0005
0006
0007        =00005000                 ORG      $5000
0008
0009        =00005000        ENTRY
0010 005000 103C 0009                 MOVE.B   #9,D0           ;display message
0011 005004 41F9 0000 50A1            LEA      MSG,A0
0012 00500A 4E40                      TRAP     #0
0013
0014 00500C 123C 000B                 MOVE.B   #10+1,D1        ;get decimal strings
0015 005010 103C 000A                 MOVE.B   #$A,D0
0016 005014 41F9 0000 5094            LEA      LIN_BUF,A0
0017 00501A 4E40                      TRAP     #0
0018
0019 00501C 615A                      BSR      CRLF_OUT
0020
0021                         *                /* convert */
0022
0023 00501E 4879 0000 5096            PEA      STR_BUF         ;push string address
0024 005024 1F28 0001                 MOVE.B   1(A0),-(SP)     ;push string length
0025 005028                  BREAK0
0026 005028 610C                      BSR      ADEC_BIN
0027 00502A                  BREAK1
0028 00502A 222F 0002                 MOVE.L   2(SP),D1        ;receive data
0029 00502E 5C8F                      ADDQ.L   #6,SP           ;adjust stack pointer
0030 005030                  BREAK2
0031 005030 103C 0000                 MOVE.B   #0,D0           ;return to system
```

```
0032 005034 4E40               TRAP    #0
0033
0034
0035                    *          ------------------------------------------
0036                    *          convert ascii decimal strings into binary
0037                    *
0038                    *                D0 : loop counter
0039                    *                D1 : work register for converting
0040                    *                D2 : work register for reading
0041                    *                D3 : work regsiter
0042                    *                A0 : string pointer
0043                    *          ------------------------------------------
0044
0045                           EVEN
0046                    ADEC_BIN
0047 005036 40E7               MOVE.W  SR,-(SP)
0048 005038 4E56 0000          LINK    A6,#0
0049 00503C 48E7 F080          MOVEM.L D0-D3/A0,-(SP)
0050
0051 005040 4280               CLR.L   D0              ;loop counter
0052 005042 4281               CLR.L   D1              ;converting register
0053 005044 4282               CLR.L   D2              ;reading register
0054
0055 005046 102E 000A          MOVE.B  10(A6),D0       ;set loop counter
0056 00504A 206E 000C          MOVEA.L 12(A6),A0       ;set address pointer
0057 00504E 5340               SUBQ.W  #1,D0           ;adjust loop counter
0058 005050 263C 0000 000A     MOVE.L  #10,D3
0059 005056             XDEC_LP
0060 005056 E389               LSL.L   #1,D1           ;D1=D1*10
0061 005058 2601               MOVE.L  D1,D3
0062 00505A E589               LSL.L   #2,D1
0063 00505C D283               ADD.L   D3,D1
0064
0065 00505E 1418               MOVE.B  (A0)+,D2        ;pickup strings ('0' to '9')
0066 005060 0482 0000 0030     SUBI.L  #$30,D2         ;convert 0 to 9
0067 005066 D282               ADD.L   D2,D1
0068 005068 51C8 FFEC          DBRA    D0,XDEC_LP
0069 00506C
0070 00506C 2D41 000C          MOVE.L  D1,12(A6)       ;return arg.
0071
0072 005070 4CDF 010F          MOVEM.L (SP)+,D0-D3/A0
0073 005074 4E5E               UNLK    A6
0074 005076 4E77               RTR
0075
0076                    *
0077                    * output CR/LF
0078                    *
0079 005078             CRLF_OUT
0080 005078 40E7               MOVE.W  SR,-(SP)        ;save registers
0081 00507A 48E7 C000          MOVEM.L D0-D1,-(SP)
0082
0083 00507E 103C 0002          MOVE.B  #2,D0           ;cr/lf
0084 005082 123C 000D          MOVE.B  #$0D,D1
0085 005086 4E40               TRAP    #0
0086 005088 123C 000A          MOVE.B  #$0A,D1
0087 00508C 4E40               TRAP    #0
0088
0089 00508E 4CDF 0003          MOVEM.L (SP)+,D0-D1     ;return registers
0090 005092 4E77               RTR
0091
0092                    *
0093                    * string area
0094                    *
0095                           EVEN
0096 005094 0B          LIN_BUF DC.B   10+1
0097 005095 =00000001          DS.B    1
0098 005096 =0000000B  STR_BUF DS.B   10+1
0099
0100 0050A1 4E75 6D62 6572  MSG  DC.B  "Number ? _",'$'
            203F 205F 24
0101
0102
0103        =00005000          END     ENTRY
```

第2部 プログラム

4 10進文字列をBCD表現へ変換する(1)

◉サンプルプログラム ［ABCD＿BCD］

■動作

\`0´～\`9´の文字セットから構成される文字列の先頭アドレスとその長さをスタックへプッシュし，ABCD＿BCDを呼び出す．結果は常にロング・ワードで返されるので，長さが8文字に達しなかった場合でも上位に＄0を満たすように処理される．

■解法

10進文字を内部BCD表現に変換するには＄30を減じればよい．あとは結果を左へ桁移動しながら文字列長だけ繰り返す．

■各行の意味

行10～19：これまでと同様文字列の取り込みを行う部分である．

行57～62：変換ループ

アプリケーション・ヒント

実際には10進文字だけが入力されるとは限らないので，本ルーチンを呼び出す前にトラップすべきである．

リスト［ABCD＿BCD］

```
LINE ADDR. CODE/VALUE        LABEL    OP      OPERAND        COMMENT

0004                                  NAM     ABCD_BCD.A68
0005
0006
0007        =00005000                 ORG     $5000
0008
0009        =00005000        ENTRY
0010 005000 103C 0009                 MOVE.B  #9,D0          ;display message
0011 005004 41F9 0000 5093            LEA     MSG,A0
0012 00500A 4E40                      TRAP    #0
0013
0014 00500C 123C 0009                 MOVE.B  #8+1,D1        ;get BCD strings
0015 005010 103C 000A                 MOVE.B  #$A,D0
0016 005014 41F9 0000 5088            LEA     LIN_BUF,A0
0017 00501A 4E40                      TRAP    #0
0018
0019 00501C 614E                      BSR     CRLF_OUT
0020
0021                         *               /* convert */
0022
0023 00501E 4879 0000 508A            PEA     STR_BUF        ;push string address
0024 005024 1F28 0001                 MOVE.B  1(A0),-(SP)    ;push string length
0025 005028                  BREAK0
0026 005028 610C                      BSR     ABCD_BCD
0027 00502A                  BREAK1
0028 00502A 222F 0002                 MOVE.L  2(SP),D1       ;receive data
0029 00502E 5C8F                      ADDQ.L  #6,SP          ;adjust stack pointer
0030 005030                  BREAK2
0031 005030 103C 0000                 MOVE.B  #0,D0          ;return to system
0032 005034 4E40                      TRAP    #0
0033
0034
0035                         *               ---------------------------------------------
0036                         *                convert ascii decimal strings into BCD data
0037                         *
0038                         *                       D0 : loop counter
0039                         *                       D1 : work register for converting
0040                         *                       D2 : work register for reading
0041                         *                       A0 : string pointer
0042                         *               ---------------------------------------------
0043
```

```
0044                                    EVEN
0045                          ABCD_BCD
0046 005036 40E7                        MOVE.W  SR,-(SP)
0047 005038 4E56 0000                   LINK    A6,#0
0048 00503C 48E7 E080                   MOVEM.L D0-D2/A0,-(SP)
0049
0050 005040 4280                        CLR.L   D0              ;loop counter
0051 005042 4281                        CLR.L   D1              ;converting register
0052 005044 4282                        CLR.L   D2              ;reading register
0053
0054 005046 102E 000A                   MOVE.B  10(A6),D0       ;set loop counter
0055 00504A 206E 000C                   MOVEA.L 12(A6),A0       ;set address pointer
0056 00504E 5340                        SUBQ.W  #1,D0           ;adjust loop counter
0057 005050                   XDEC_LP
0058 005050 E989                        LSL.L   #4,D1           ;logical shift left for 4 bit
0059 005052 1418                        MOVE.B  (A0)+,D2        ;pickup strings ('0' to '9')
0060 005054 0482 0000 0030              SUBI.L  #$30,D2         ;convert 0 to 9
0061 00505A 8282                        OR.L    D2,D1
0062 00505C 51C8 FFF2                   DBRA    D0,XDEC_LP
0063 005060
0064 005060 2D41 000C                   MOVE.L  D1,12(A6)       ;return arg.
0065
0066 005064 4CDF 0107                   MOVEM.L (SP)+,D0-D2/A0
0067 005068 4E5E                        UNLK    A6
0068 00506A 4E77                        RTR
0069
0070                          *
0071                          * output CR/LF
0072                          *
0073 00506C                   CRLF_OUT
0074 00506C 40E7                        MOVE.W  SR,-(SP)        ;save registers
0075 00506E 48E7 C000                   MOVEM.L D0-D1,-(SP)
0076
0077 005072 103C 0002                   MOVE.B  #2,D0           ;cr/lf
0078 005076 123C 000D                   MOVE.B  #$0D,D1
0079 00507A 4E40                        TRAP    #0
0080 00507C 123C 000A                   MOVE.B  #$0A,D1
0081 005080 4E40                        TRAP    #0
0082
0083 005082 4CDF 0003                   MOVEM.L (SP)+,D0-D1     ;return registers
0084 005086 4E77                        RTR
0085
0086                          *
0087                          * string area
0088                          *
0089                                    EVEN
0090 005088 09                LIN_BUF   DC.B    8+1
0091 005089 =00000001                   DS.B    1
0092 00508A =00000009         STR_BUF   DS.B    8+1
0093
0094 005093 4243 4420 4E75    MSG       DC.B    "BCD Number ? _",'$'
            6D62 6572 203F
            205F 24
0095
0096
0097        =00005000                   END     ENTRY
```

第2部 プログラム

5 10進文字列をBCD表現へ変換する(2)

◉サンプルプログラム [BBCD__BCD]

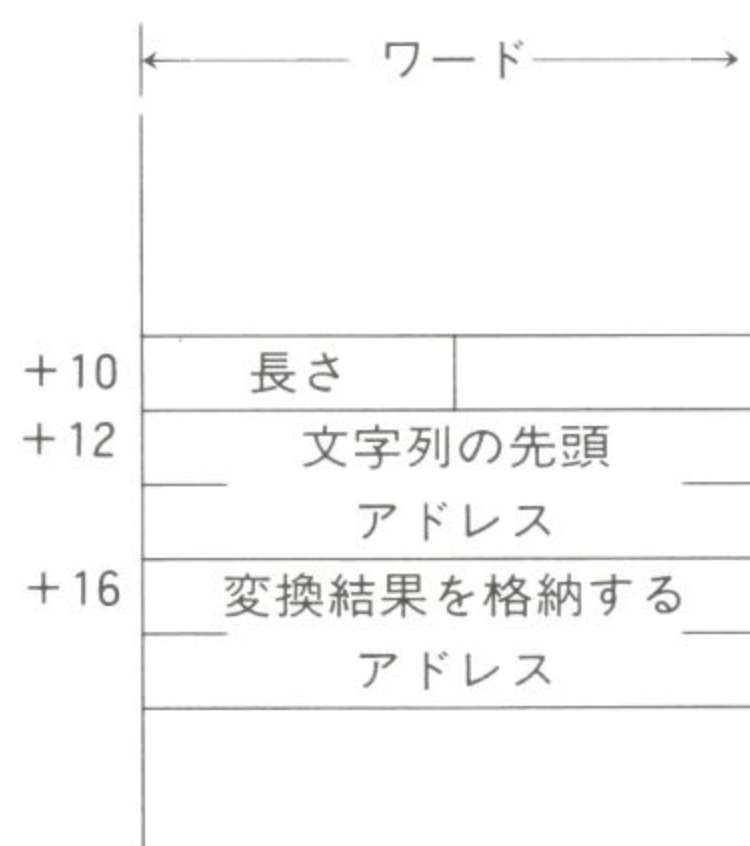
「ABCD__BCD」では8桁までしか扱えませんでしたが，ここでは変換値を格納する場所をメモリ上に確保し桁数の制約をなくしました．

■動作

変換値の格納アドレス，`0´～`9´の文字セットから構成される文字列の先頭アドレス，文字列長を順にスタックへプッシュし,BBCD__BCDを呼び出す．結果は指定した場所から順に格納される．

■解法

結果の格納先をメモリ上に変更した他は「ABCD__BCD」とまったく同様である．

図2.51 BBCD__BCDとスタック

リスト[BBCD__BCD]

```
LINE ADDR. CODE/VALUE         LABEL    OP       OPERAND        COMMENT

0004                                   NAM      BBCD_BCD.A68
0005
0006
0007        =00005000                  ORG      $5000
0008
0009        =00005000         ENTRY
0010 005000 103C 0009                  MOVE.B   #9,D0          ;display message
0011 005004 41F9 0000 50D5             LEA      MSG,A0
0012 00500A 4E40                       TRAP     #0
0013
0014 00500C 123C 0021                  MOVE.B   #32+1,D1       ;get BCD strings
0015 005010 103C 000A                  MOVE.B   #$A,D0
0016 005014 41F9 0000 5092             LEA      LIN_BUF,A0
0017 00501A 4E40                       TRAP     #0
0018
0019 00501C 6158                       BSR      CRLF_OUT
0020
0021                          *                 /* convert */
0022
0023 00501E 4879 0000 50B5             PEA      CONV_BUF       ;push load address
0024 005024 4879 0000 5094             PEA      STR_BUF        ;push string address
0025 00502A 1F28 0001                  MOVE.B   1(A0),-(SP)    ;push string length
0026 00502E                   BREAK0
0027 00502E 610C                       BSR      BBCD_BCD
0028 005030                   BREAK1
0029 005030 DFFC 0000 000A             ADDA.L   #10,SP         ;adjust stack pointer
0030 005036                   BREAK2
0031 005036 103C 0000                  MOVE.B   #0,D0          ;return to system
0032 00503A 4E40                       TRAP     #0
```

```
0033
0034
0035                         *           --------------------------------------------
0036                         *           convert ascii decimal strings into BCD data
0037                         *
0038                         *                  D0 : loop counter
0039                         *                  D1 : work register
0040                         *                  D2 : work register
0041                         *                  A0 : string pointer
0042                         *                  A1 : BCD data load address
0043                         *           --------------------------------------------
0044
0045                                 EVEN
0046                         BBCD_BCD
0047 00503C 40E7                     MOVE.W  SR,-(SP)
0048 00503E 4E56 0000                LINK    A6,#0
0049 005042 48E7 E0C0                MOVEM.L D0-D2/A0-A1,-(SP)
0050
0051 005046 4280                     CLR.L   D0              ;clear loop counter
0052 005048 102E 000A                MOVE.B  10(A6),D0       ;set loop counter
0053 00504C 206E 000C                MOVEA.L 12(A6),A0       ;set address pointer for read
0054 005050 226E 0010                MOVEA.L 16(A6),A1       ;set address pointer for write
0055
0056 005054 E248                     LSR.W   #1,D0           ;obtaine byte count,must be even
0057 005056 5340                     SUBQ.W  #1,D0           ;adjust loop counter
0058 005058                  BDEC_LP
0059 005058 1218                     MOVE.B  (A0)+,D1        ;convert upper nibble
0060 00505A 0401 0030                SUBI.B  #$30,D1
0061 00505E E909                     LSL.B   #4,D1
0062
0063 005060 1418                     MOVE.B  (A0)+,D2        ;convert lower nibble
0064 005062 0402 0030                SUBI.B  #$30,D2
0065 005066 8202                     OR.B    D2,D1
0066
0067 005068 12C1                     MOVE.B  D1,(A1)+        ;load BCD data
0068 00506A 51C8 FFEC                DBRA    D0,BDEC_LP
0069
0070 00506E 4CDF 0307                MOVEM.L (SP)+,D0-D2/A0-A1
0071 005072 4E5E                     UNLK    A6
0072 005074 4E77                     RTR
0073
0074                         *
0075                         * output CR/LF
0076                         *
0077 005076                  CRLF_OUT
0078 005076 40E7                     MOVE.W  SR,-(SP)        ;save registers
0079 005078 48E7 C000                MOVEM.L D0-D1,-(SP)
0080
0081 00507C 103C 0002                MOVE.B  #2,D0           ;cr/lf
0082 005080 123C 000D                MOVE.B  #$0D,D1
0083 005084 4E40                     TRAP    #0
0084 005086 123C 000A                MOVE.B  #$0A,D1
0085 00508A 4E40                     TRAP    #0
0086
0087 00508C 4CDF 0003                MOVEM.L (SP)+,D0-D1     ;return registers
0088 005090 4E77                     RTR
0089
0090                         *
0091                         * string area
0092                         *
0093                                 EVEN
0094 005092 20               LIN_BUF DC.B    31+1
0095 005093 =00000001                DS.B    1
0096 005094 =00000021        STR_BUF DS.B    32+1
0097
0098 0050B5 =00000020        CONV_BUF DS.B   32
0099
0100 0050D5 4243 4420 4E75   MSG     DC.B    "BCD Number ? _",'$'
            6D62 6572 203F
            205F 24
0101
0102
0103        =00005000                END     ENTRY
```

6 バイナリ表現をビット列の文字列へ変換する

◉サンプルプログラム ［BIN__ABIT］

■動作

変換した文字（`0´か`1´のいずれか）の格納アドレス，変換を必要とするバイナリ値，変換文字数を順にスタックへプッシュし，BIN__ABITを呼び出す．

結果は指定アドレスへ格納されるので，その内容（文字列）をスクリーンへ表示して確認する（すべてのサイズをサポートしている）．

■解法

データを左へ１ビットシフトするとその内容がCCRのキャリへ反映されるので，それに応じて`0´または`1´の文字コードを生成し，これを指定場所へ格納すればよい．何ビット処理すべきかはメインから与えられるので，これでループ制御する。

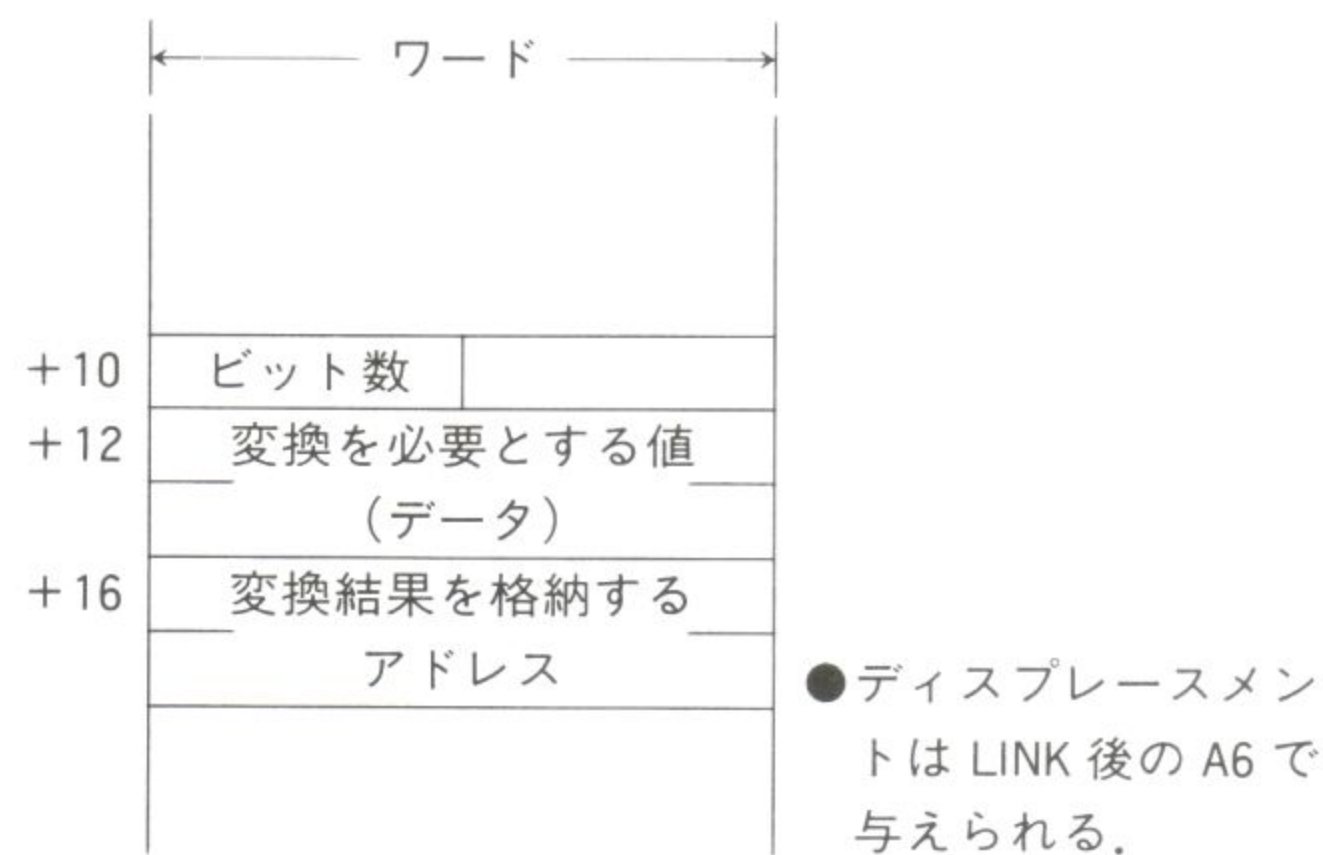
■各行の意味

行15～64：メイン側ルーチンであり，32/16/8ビットの各サイズのテスト・データで確認をしている．

行77～130：メインから渡された引数に応じてビット列へ変換するBIN__ABINが位置している．

図2.52 BIN__ABITとスタック

	ワード
+10	ビット数
+12	変換を必要とする値
	（データ）
+16	変換結果を格納する
	アドレス

●ディスプレースメントはLINK後のA6で与えられる．

リスト［BIN__ABIT］

```
LINE ADDR. CODE/VALUE          LABEL    OP       OPERAND          COMMENT

0004                                    NAM      BIN_ABIT.A68
0005
0006           =AAAAAAAA        DATA_32  EQU      $AAAAAAAA        ;define test data
0007           =0000AAAA        DATA_16  EQU      $AAAA
0008           =000000AA        DATA_8   EQU      $AA
0009
0010
0011           =00005000                 ORG      $5000
0012
0013                            *                 /* test 32 bit */
0014           =00005000        ENTRY
0015 005000 4879 0000 50D0               PEA      STR_32           ;push string load address
0016 005006 2F3C AAAA AAAA               MOVE.L   #DATA_32,-(SP)   ;push data for converting
0017 00500C 1F3C 0020                    MOVE.B   #32,-(SP)        ;push bit number
0018 005010                     BREAK0
0019 005010 6156                         BSR      BIN_ABIT
```

```
0020
0021                           *          /* display 32 bit */
0022
0023 005012 206F 0006                  MOVEA.L 6(SP),A0        ;pop load address pointer
0024 005016 DFFC 0000 000A             ADDA.L  #10,SP          ;adjust stack pointer
0025 00501C                    BREAK1
0026 00501C 6142                        BSR     DSP_MSG
0027
0028                           *          /* test 16 bit */
0029
0030 00501E 4879 0000 50F3             PEA     STR_16          ;push string load address
0031 005024 2F3C 0000 AAAA             MOVE.L  #DATA_16,-(SP)  ;push data for converting
0032 00502A 1F3C 0010                  MOVE.B  #16,-(SP)       ;push bit number
0033 00502E                    BREAK2
0034 00502E 6138                        BSR     BIN_ABIT
0035                           *          /* display 16 bit */
0036
0037 005030 206F 0006                  MOVEA.L 6(SP),A0        ;pop load address pointer
0038 005034 DFFC 0000 000A             ADDA.L  #10,SP          ;adjust stack pointer
0039 00503A                    BREAK3
0040 00503A 6124                        BSR     DSP_MSG
0041
0042                           *          /* test 8 bit */
0043
0044 00503C 4879 0000 5106             PEA     STR_8           ;push string load address
0045 005042 2F3C 0000 00AA             MOVE.L  #DATA_8,-(SP)   ;push data for converting
0046 005048 1F3C 0008                  MOVE.B  #8,-(SP)        ;push bit number
0047 00504C                    BREAK4
0048 00504C 611A                        BSR     BIN_ABIT
0049                           *          /* display 8 bit */
0050
0051 00504E 206F 0006                  MOVEA.L 6(SP),A0        ;pop load address pointer
0052 005052 DFFC 0000 000A             ADDA.L  #10,SP          ;adjust stack pointer
0053 005058                    BREAK5
0054 005058 6106                        BSR     DSP_MSG
0055
0056 00505A 103C 0000                  MOVE.B  #0,D0           ;return to system
0057 00505E 4E40                        TRAP    #0
0058                           *
0059                           * display sub
0060                           *
0061 005060                    DSP_MSG
0062 005060 103C 0009                  MOVE.B  #9,D0           ;function 9
0063 005064 4E40                        TRAP    #0
0064 005066 4E75                        RTS
0065
0066                           *          ----------------------------------------
0067                           *           convert binary into asc binary strings
0068                           *
0069                           *          D0 : lop counter
0070                           *          D1 : data from main (argument)
0071                           *          D2 : work register
0072                           *          A0 : load address pointer
0073                           *          ----------------------------------------
0074
0075                                    EVEN
0076                           BIN_ABIT
0077 005068 40E7                        MOVE.W  SR,-(SP)
0078 00506A 4E56 0000                  LINK    A6,#0
0079 00506E 48E7 E080                  MOVEM.L D0-D2/A0,-(SP)
0080
0081 005072 4280                        CLR.L   D0
0082 005074 206E 0010                  MOVEA.L 16(A6),A0       ;set write buffer address pointer
0083 005078 222E 000C                  MOVE.L  12(A6),D1       ;pickup data for convert
0084 00507C 102E 000A                  MOVE.B  10(A6),D0       ;set bit size for loop control
0085 005080 5340                        SUBQ.W  #1,D0           ;adjust loop counter
0086
0087 005082 0C00 0007                  CMPI.B  #7,D0           ;test converting bit size
0088 005086 6730                        BEQ     WIDTH_8
0089 005088 0C00 000F                  CMPI.B  #15,D0
0090 00508C 6718                        BEQ     WIDTH_16
0091 00508E 0C00 001F                  CMPI.B  #31,D0
0092 005092 6634                        BNE     ESC_ABIT
0093
0094                           *          /* convert 32 bit into 32 asc strings */
0095 005094                    WIDTH_32
0096 005094 143C 0030                  MOVE.B  #'0',D2         ;initialize work register
0097 005098 E389                        LSL.L   #1,D1
0098 00509A 6402                        BCC     WR_STR_32
0099 00509C 5202                        ADDQ.B  #1,D2
0100 00509E                    WR_STR_32
0101 00509E 10C2                        MOVE.B  D2,(A0)+        ;load '0' or '1'
0102 0050A0 51C8 FFF2                  DBRA    D0,WIDTH_32
```

```
0103 0050A4 6022                   BRA     ESC_ABIT
0104
0105                      *                /* convert 16 bin into 16 asc strings */
0106 0050A6               WIDTH_16
0107 0050A6 143C 0030              MOVE.B  #'0',D2
0108 0050AA E349                   LSL.W   #1,D1
0109 0050AC 6402                   BCC     WR_STR_16
0110 0050AE 5202                   ADDQ.B  #1,D2
0111 0050B0               WR_STR_16
0112 0050B0 10C2                   MOVE.B  D2,(A0)+
0113 0050B2 51C8 FFF2              DBRA    D0,WIDTH_16
0114 0050B6 6010                   BRA     ESC_ABIT
0115
0116                      *                /* convert 8 bit into asc strings */
0117 0050B8               WIDTH_8
0118 0050B8 143C 0030              MOVE.B  #'0',D2
0119 0050BC E309                   LSL.B   #1,D1

PAGE 2  BIN_ABIT.PRN  January 18, 1986  23:53:40
0120 0050BE 6402                   BCC     WR_STR_8
0121 0050C0 5202                   ADDQ.B  #1,D2
0122 0050C2               WR_STR_8
0123 0050C2 10C2                   MOVE.B  D2,(A0)+
0124 0050C4 51C8 FFF2              DBRA    D0,WIDTH_8
0125
0126                      *                /* return job */
0127 0050C8               ESC_ABIT
0128 0050C8 4CDF 0107              MOVEM.L (SP)+,D0-D2/A0
0129 0050CC 4E5E                   UNLK    A6
0130 0050CE 4E77                   RTR
0131                      *
0132                      * string area
0133                      *
0134
0135 0050D0 =00000020     STR_32   DS.B    32
0136 0050F0 0D0A 24                DC.B    $0D,$0A,'$'
0137
0138 0050F3 =00000010     STR_16   DS.B    16
0139 005103 0D0A 24                DC.B    $0D,$0A,'$'
0140
0141 005106 =00000008     STR_8    DS.B    8
0142 00510E 0D0A 24                DC.B    $0D,$0A,'$'
0143
0144
0145        =00005000              END     ENTRY
```

7 バイナリ表現を16進文字列へ変換する

◉サンプルプログラム ［BIN__AHEX］

■動作

変換された文字の格納アドレス，変換値，変換サイズ（ロング・ワード：8，ワード：4，バイト：2）を順にスタックへプッシュし，BIN__AHEXを呼び出す．

本例ではあらかじめ用意しておいた各データを順に変換し，その結果をスクリーンへ表示して確認している．

■解法

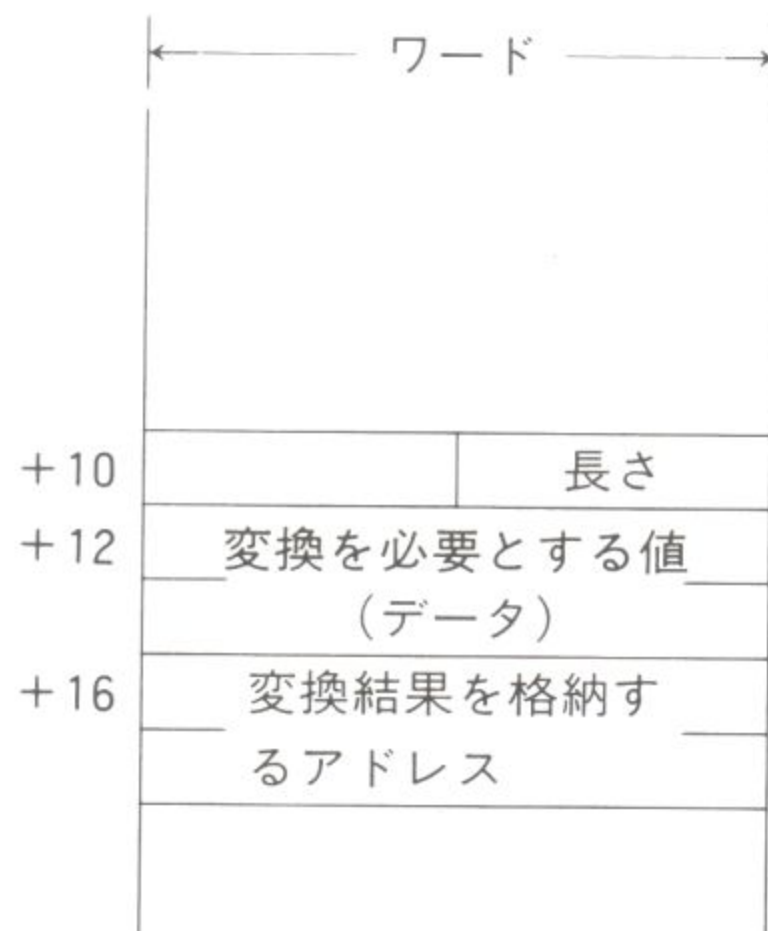
メモリ上の1バイトを文字で表現するには2文字を必要とするので，たとえばある番地の内容が＄A0ならば，`A´という文字コードと`0´という文字コードに変換しなくてはならない．＄0を`0´という文字コードへ変換するには＄30を加算すればよいが，`9´と`A´との間は連続していないので，さらにこのための不足分を加算して`A´～`F´へ変換すればよい．

■各行の意味

行15～60　：各データ・サイズを16進文字列に変換し，確認のためにスクリーンへ表示する．

行72～150　：ここでの基本は133～150に位置する1バイトを16進文字に変換するルーチンB__HEXで，ワードやロング・ワードの処理もこれを呼び出している．

図2.53　BIN__AHEXとスタック

	ワード	
＋10		長さ
＋12	変換を必要とする値（データ）	
＋16	変換結果を格納するアドレス	

●メイン側では，MOVE.W　＃8，－(SP)のようにしているので，長さは図の位置へ格納されてサブルーチンへくる．
もし，
　　MOVE.B　＃8，－(SP)
のようにすれば，オフセットは＋10の位置である．
いずれにしても SP(スタック・ポインタ)は－2だけ変更される．
●ディスプレースメントは LINK 後の A6で与えられる．

リスト［BIN__AHEX］

```
LINE ADDR. CODE/VALUE          LABEL     OP        OPERAND          COMMENT

0004                                     NAM       BIN_AHEX.A68
0005
0006        =AA55AA55          LWORD_DAT EQU       $AA55AA55        ;define test data
0007        =0000AA55          WORD_DAT  EQU       $AA55
0008        =000000A5          BYTE_DAT  EQU       $A5
0009
0010
0011        =00005000                    ORG       $5000
0012
0013                           *                  /* test long word data */
0014        =00005000          ENTRY
0015 005000 4879 0000 50D6               PEA       LWORD_STR        ;push string load address
0016 005006 2F3C AA55 AA55               MOVE.L    #LWORD_DAT,-(SP) ;push data for converting
```

```
0017 00500C 3F3C 0008                    MOVE.W  #8,-(SP)        ;push convert size
0018 005010                      BREAK0
0019 005010 6156                         BSR     BIN_AHEX
0020
0021 005012 206F 0006                    MOVEA.L 6(SP),A0        ;pop load address pointer
0022 005016 DFFC 0000 000A               ADDA.L  #10,SP          ;adjust stack pointer
0023 00501C                      BREAK1
0024 00501C 6142                         BSR     DSP_MSG         ; /* display long word */
0025
0026                             *               /* test word data */
0027
0028 00501E 4879 0000 50E1               PEA     WORD_STR        ;push string load address
0029 005024 2F3C 0000 AA55               MOVE.L  #WORD_DAT,-(SP) ;push data for converting
0030 00502A 3F3C 0004                    MOVE.W  #4,-(SP)        ;push convert size
0031 00502E                      BREAK2
0032 00502E 6138                         BSR     BIN_AHEX
0033
0034 005030 206F 0006                    MOVEA.L 6(SP),A0        ;pop load address pointer
0035 005034 DFFC 0000 000A               ADDA.L  #10,SP          ;adjust stack pointer
0036 00503A                      BREAK3
0037 00503A 6124                         BSR     DSP_MSG         ;/* display word */
0038
0039                             *               /* test byte data */
0040
0041 00503C 4879 0000 50E8               PEA     BYTE_STR        ;push string load address
0042 005042 2F3C 0000 00A5               MOVE.L  #BYTE_DAT,-(SP) ;push data for converting
0043 005048 3F3C 0002                    MOVE.W  #2,-(SP)        ;push convert size
0044 00504C                      BREAK4
0045 00504C 611A                         BSR     BIN_AHEX
0046
0047 00504E 206F 0006                    MOVEA.L 6(SP),A0        ;pop load address pointer
0048 005052 DFFC 0000 000A               ADDA.L  #10,SP          ;adjust stack pointer
0049 005058                      BREAK5
0050 005058 6106                         BSR     DSP_MSG         ;/* display byte */
0051
0052 00505A 103C 0000                    MOVE.B  #0,D0           ;return to system
0053 00505E 4E40                         TRAP    #0
0054                             *
0055                             * display sub
0056                             *
0057 005060                      DSP_MSG
0058 005060 103C 0009                    MOVE.B  #9,D0           ;function 9
0059 005064 4E40                         TRAP    #0
0060 005066 4E75                         RTS
0061
0062                             *               ----------------------------------------
0063                             *                convert binary into asc hex strings
0064                             *
0065                             *               D0 : convert size (8/4/2)
0066                             *               D1 : data from main (argument)
0067                             *               A0 : load address pointer
0068                             *               ----------------------------------------
0069
0070                                     EVEN
0071                             BIN_AHEX
0072 005068 40E7                         MOVE.W  SR,-(SP)
0073 00506A 4E56 0000                    LINK    A6,#0
0074 00506E 48E7 C080                    MOVEM.L D0-D1/A0,-(SP)
0075
0076 005072 206E 0010                    MOVEA.L 16(A6),A0       ;set write buffer address pointer
0077 005076 222E 000C                    MOVE.L  12(A6),D1       ;pickup data for convert
0078 00507A 302E 000A                    MOVE.W  10(A6),D0       ;set bit size for loop control
0079
0080 00507E 0C40 0002                    CMPI.W  #2,D0           ;test converting bit size
0081 005082 671C                         BEQ     BYTE_HEX
0082 005084 0C40 0004                    CMPI.W  #4,D0
0083 005088 6712                         BEQ     WORD_HEX
0084 00508A 0C40 0008                    CMPI.W  #8,D0
0085 00508E 6612                         BNE     ESC_AHEX
0086
0087                             *               /* convert long word into hex strings /
0088 005090                      LWORD_HEX
0089 005090 2F01                         MOVE.L  D1,-(SP)        ;save D1.L
0090 005092 4841                         SWAP    D1
0091 005094 6114                         BSR     W_HEX
0092
0093 005096 221F                         MOVE.L  (SP)+,D1        ;return D1.L
0094 005098 6110                         BSR     W_HEX
0095 00509A 6006                         BRA     ESC_AHEX
0096
0097                             *               /* convert word into hex strings */
0098 00509C                      WORD_HEX
0099 00509C 610C                         BSR     W_HEX
```

```
0100 00509E 6002                     BRA     ESC_AHEX
0101
0102                          *              /* convert byte into hex strings */
0103 0050A0                   BYTE_HEX
0104 0050A0 6114                     BSR     B_HEX
0105
0106                          *              /* return job */
0107 0050A2                   ESC_AHEX
0108 0050A2 4CDF 0103                MOVEM.L (SP)+,D0-D1/A0
0109 0050A6 4E5E                     UNLK    A6
0110 0050A8 4E77                     RTR
0111
0112                          *              -------------------------------------
0113                          *               convert word into ascii hex strings
0114                          *
0115                          *                      D1.W = data
0116                          *                      A0   = load address
0117                          *              -------------------------------------
0118 0050AA                   W_HEX
0119 0050AA 3F01                     MOVE.W  D1,-(SP)        ;save D1.W

PAGE 2  BIN_AHEX.PRN  January 20, 1986  22:16:26
0120 0050AC E049                     LSR.W   #8,D1
0121 0050AE 6106                     BSR     B_HEX
0122
0123 0050B0 321F                     MOVE.W  (SP)+,D1        ;return D1.W
0124 0050B2 6102                     BSR     B_HEX
0125 0050B4 4E75                     RTS
0126
0127                          *              -------------------------------------
0128                          *               convert byte into ascii hex strings
0129                          *
0130                          *                      D1.B = data
0131                          *                      A0   = load address
0132                          *              -------------------------------------
0133 0050B6                   B_HEX
0134 0050B6 1F01                     MOVE.B  D1,-(SP)        ;save D1
0135 0050B8 0201 00F0                ANDI.B  #$F0,D1         ;pickup upper nibble
0136 0050BC E809                     LSR.B   #4,D1
0137 0050BE 6106                     BSR     PUT_HEX
0138
0139 0050C0 121F                     MOVE.B  (SP)+,D1        ;return D1
0140 0050C2 0201 000F                ANDI.B  #$0F,D1         ;pickup lower nibble
0141
0142                          *              /* convert nibble into ascii hex strings,then load to memory */
0143 0050C6                   PUT_HEX
0144 0050C6 0601 0030                ADDI.B  #$30,D1         ;get '0' to 'F'
0145 0050CA 0C01 003A                CMPI.B  #$3A,D1
0146 0050CE 6B02                     BMI     LD_HEX          ;jump LD_HEX,already '0' to '9'
0147 0050D0 5E01                     ADDQ.B  #7,D1           ;get 'A' to 'F'
0148 0050D2                   LD_HEX
0149 0050D2 10C1                     MOVE.B  D1,(A0)+
0150 0050D4 4E75                     RTS
0151
0152                          *
0153                          * string area
0154                          *
0155
0156 0050D6 =00000008         LWORD_STR DS.B  8
0157 0050DE 0D0A 24                  DC.B    $0D,$0A,'$'
0158
0159 0050E1 =00000004         WORD_STR DS.B   4
0160 0050E5 0D0A 24                  DC.B    $0D,$0A,'$'
0161
0162 0050E8 =00000002         BYTE_STR DS.B   2
0163 0050EA 0D0A 24                  DC.B    $0D,$0A,'$'
0164
0165 0050ED 00                Z       DC.B    0
0166
0167
0168        =00005000                END     ENTRY
```

8 バイナリ表現を10進文字列へ変換する

◉サンプルプログラム [BIN__ADEC]

■動作

変換した文字列を格納するアドレス，D０に格納された変換データ（32ビット）をスタックへプッシュしBIN__ADECを呼び出す．文字列は10桁の10進文字列として求められ，場合によっては上位に〝０″が満たされる．

■解法

変換の方法であるが，ここでは汎用性とスピードを考慮したアルゴリズムを採用する．あらかじめメモリ上に〝10のべき″を格納したテーブルを用意し，各桁の値はこの〝べきテーブル″の内容を繰り返し減算して求める，

たとえば３桁の数値255（にひゃく　ごじゅうご）を例にとってみる．

① べきテーブルから100を取り出して減算できる回数を求める　　－ 2 －
（ここで元の値は55になっている）

② べきテーブルから10を取り出して減算できる回数を求める　　－ 5 －
（ここで元の値は５になっている）

③ べきテーブルから１を取り出して同様な処理を行う　　－ 5 －

このようにして各桁が求まるので，これに＄30を加算すれば〝２″，〝５″，〝５″なる10進文字列が求まる（除算によって桁の値を求めることもできるが，32ビットの除算命令はサポートされていないし，変換スピードも遅い）．

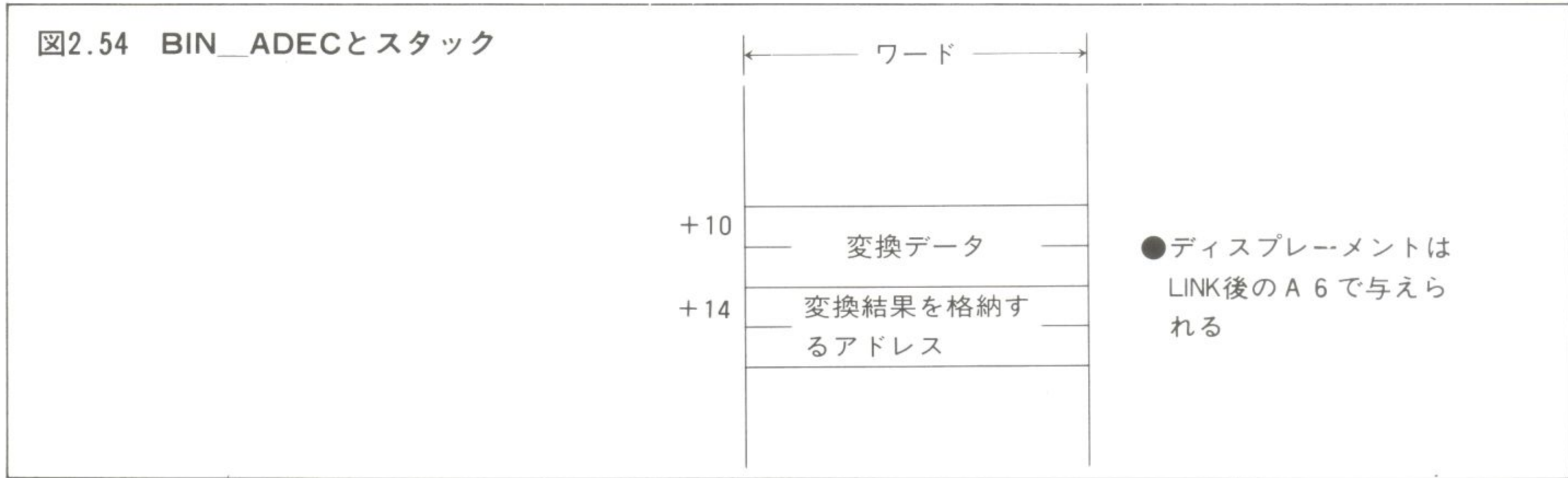
■各行の意味

行９～12：引数をスタックへプッシュしBIN__ADECを呼び出す．サブルーチンからのリターン情報は考えていないのでSPへの加算でSPを復元している．

行22～31：べきテーブル（10桁分）

以下はメインから渡された値を10進文字に変換し，指定アドレスへ格納するルーチンである．

行46～51：先に説明した〝何回引けるか″をカウントしている．

図2.54　BIN__ADECとスタック

リスト[BIN__ADEC]

```
LINE ADDR. CODE/VALUE         LABEL    OP       OPERAND        COMMENT

0005
0007        =00005000                  ORG      $5000
0008
0009 005000 4879 0000 5078             PEA      STR_BUF
0010 005006 2F00                       MOVE.L   D0,-(SP)
0011 005008 4EB9 0000 503C             JSR      BIN_ADEC
0012 00500E 508F                       ADDQ.L   #8,SP          ;adjust stack pointer
0013 005010                   BREAK
0015 005010 7000                       MOVEQ    #0,D0
0016 005012 4E40                       TRAP     #0
0017
0018                          *        ------------------------------------
0019                          *        [sub] convert binary into BCD string
0020                          *        ------------------------------------
0021
0022 005014 3B9A CA00         EXP_TBL  DC.L     1000000000
0023 005018 05F5 E100                  DC.L     100000000
0024 00501C 0098 9680                  DC.L     10000000
0025 005020 000F 4240                  DC.L     1000000
0026 005024 0001 86A0                  DC.L     100000
0027 005028 0000 2710                  DC.L     10000
0028 00502C 0000 03E8                  DC.L     1000
0029 005030 0000 0064                  DC.L     100
0030 005034 0000 000A                  DC.L     10
0031 005038 0000 0001                  DC.L     1
0032
0033 00503C                   BIN_ADEC
0034 00503C 40E7                       MOVE.W   SR,-(SP)
0035 00503E 4E56 0000                  LINK     A6,#0
0036 005042 48E7 F0C0                  MOVEM.L  D0-D3/A0-A1,-(SP)
0037
0038 005046 202E 000A                  MOVE.L   10(A6),D0      ;D0: data from main
0039 00504A 206E 000E                  MOVEA.L  14(A6),A0      ;A0: string load address form main
0040
0041 00504E 7209                       MOVEQ    #9,D1          ;D1: loop counter
0042 005050 43F9 0000 5014             LEA      EXP_TBL,A1     ;A1: pointer of exp_tbl
0043 005056                   EX_LOOP0
0044 005056 4202                       CLR.B    D2             ;D2: work counter
0045 005058 2619                       MOVE.L   (A1)+,D3
0046 00505A                   EX_LOOP1
0047 00505A 8643                       OR       D3,D3          ;clear carry flag
0048 00505C 9083                       SUB.L    D3,D0          ;D0=D0-D3
0049 00505E 6504                       BCS      XBCD_STR
0050 005060 5202                       ADDQ.B   #1,D2          ;count-up work register
0051 005062 60F6                       BRA      EX_LOOP1
0052 005064                   XBCD_STR
0053 005064 D083                       ADD.L    D3,D0          ;D0=D0+D3
0054 005066 D43C 0030                  ADD.B    #'0',D2        ;convert work counter into BCD string
0055 00506A 10C2                       MOVE.B   D2,(A0)+       ;load BCD string
0056 00506C 51C9 FFE8                  DBRA     D1,EX_LOOP0
0057
0058 005070 4CDF 030F                  MOVEM.L  (SP)+,D0-D3/A0-A1
0059 005074 4E5E                       UNLK     A6
0060 005076 4E77                       RTR
0061
0062
0063                          *        ----------------
0064                          *        string load area
0065                          *        ----------------
0066
0067 005078 =00000014         STR_BUF  DS.B     10*2
0068
0069                                   END
```

行53～56：求められた回数を10進文字に変換して格納する．
53行で行っている加算命令であるが，引ける回数を求めるループをぬけるためには引けなくなるまで減算を行うので，ここへエントリしたときには１回だけ余分に減算されているので，ここで必要な値を復元しているのである．

初歩的な例題から汎用サブルーチンの作成まで解説してきましたが，しつこいようですが特に以下の２点を確認しておきましょう．

■スクリーン・エディタ

もしあなたが初心者であったとしても，「初心者向け」のエディタを購入すべきではありません．エディタのコマンドというものは，２～３日も使えば「初心者ではなくなる」からです．このようなものだと思うのではなく，「好みにうるさい人」になってほしいわけです．

■スタックを介して引数の授受を行うことの大切さ

68000内には豊富なレジスタ群があり，メイン～サブルーチン間の通信はレジスタを介して行いたくなります．しかしサブルーチン内での処理を完全にブラック・ボックス化する必要があり，引数をスタック経由で授受することはきわめて当然なことです．
もちろんサブルーチン内では，

① **ローカル・エリアの確保**
② **SRの退避やMOVEM命令によるレジスタ群の退避**

などを必要に応じて行います．
重要なことはスタックへプッシュする順とその内容であり，例題でみられるように，Ｄ０の内容をプッシュしそれをサブルーチン側のＤ０で受けたとしても，メイン側とサブルーチン側とは完全に分離されていることを忘れないでほしいのです．メイン側でプッシュされる内容がＤ０ではなくＤ６であろうとＤ４であろうと問題ではないのですから….

18 モジュール別開発と市販ツール

■モジュール別開発

より高い次元の記述をするには，命令セットをいくつか組み合わせなければなりません．我々にとっては，「ある文字をプリンタへ出力したい」といった具体的な記述こそ有用なのですから，「文字の格納アドレスをアドレス・レジスタへ転送する」では，何の意味もないわけです．このように「意味のある処理」を実現するにはいくつかの命令が必要であり，複数のサブルーチンを呼び出すことになります．

「意味のある処理部」をモジュールと呼びますが，プログラミングとは，

① どのようなモジュールが必要か

② そのモジュールにはどのようなサブルーチンが必要か

というように高位から低位へプログラミング・レベルを解析します．その後要求されるすべてのサブルーチンを開発し，これらを組み合わせて1つのモジュールを完成します．

したがって1回目より2回目，2回目より3回目というように，モジュール数の増大とともに開発期間は驚異的に短縮されることになります．

■リロケータブル・アセンブラ

モジュール別開発を効果的に行うためには，リロケータブル・アセンブラと呼ばれる種類のアセンブラが必要であり，リロケータブル・アセンブラにはマクロ機能がサポートされることが常識なので，単にマクロ・アセンブラと呼ばれることもあります．

リロケータブル・アセンブラの考え方は「モジュールをリンクする」ということなので，アセンブルされたオブジェクト・ファイルはオブジェクト・リンカ（単にリンカともいう）へ入力され，リンカによってモジュールをリンクしないと実行型コードは得られません．これは不便なように思われますが，1回のアセンブル作業はモジュールを作成することなので，アセンブル回数の増大とともに高級言語レベルの開発が可能となります．

またリロケータブル・アセンブラには〝ライブラリアン〟というユーティリティが付属しています．これはモジュールの整理をするもので，モジュールとモジュールを結合して1つの新しいモジュールとしたり，不要となったモジュールを削除したりするものです．

このような次元になりますと，以下のようなプログラムを操作しなければなりません．

① **スクリーン・エディタ**

② **アセンブラ**

③ **リンカ**

④ **ライブラリアン**

■市販開発ツール

市販されている68000クロス・アセンブラですが，アメリカではIBM-PCのユーザが圧倒的であることもあり，MS-DOS（PC-DOS）上で動作するものがいくつかあります．以下の2点はいずれも十分な機能を備えており，68000のプログラム開発には強力なツールとなるでしょう．

① Quelo

② u−Series Assembler

第3部　命令の詳細

本セクションでは個別命令の詳細について解説しますが，各命令は以下のような項目から構成されています．

●命令の解説
●CCRの変化
●機械語フォーマット
●アドレッシング・モードとオブジェクト・サイズ／クロック・サイクル数

例：MOVE　<ea>,<ea>

縦はソース・オペランドで指定可能なバリエーション，横にディスティネーション・オペランドで指定可能なバリエーションが整理されており，オペランド・サイズはＢ，Ｗ，Ｌ，と明示されています．これらの交点には〝#〞と〝～〞の欄があり，

#　――　該当する命令のオブジェクト・バイト数
～　――　クロック・サイクル数

が記入されています．ソース欄とディスティネーション欄は自由に組み合わせ可能ですから，

MOVE.B (A0),D0

という命令は，Ａ０でポイントされるメモリ内に存在するバイト・データをＤ０へ転送するもので，オブジェクト・サイズは2，クロック・サイクル数は8であることがわかります．

通常の用途ではオブジェクト・サイズやクロック数を気にしながらプログラミングすることはあり得ませんが，ソース・オペランドとディスティネーション・オペランドとの対応を一覧表にしたかったので，ついでにこれらのパラメータを記入してあります．

表を御覧の通り，たった１つの転送命令でも大変な数の命令群が用意されていることを御理解頂けると思いますが，もちろんAnやDnは一般形であり，Ａ０～Ａ７，Ｄ０～Ｄ７のすべてを意味します．

なお命令によっては，アドレッシングの表が一方的に縦方向や横方向に成長するものがあり，レイアウトに苦心しました．そこで項目の順が前後することもありますが，できるだけページ間へまたがることのないように配慮し，必要な情報を的確かつ素早く取り出せるようにしたつもりです．

1 ◉MOVE [MOVE data 転送]

MOVE {.B/.W/.L} <ea>, <ea>	X	N	Z	V	C
	—	*	*	0	0

解説 汎用のデータ転送命令で，左側（ソース・オペランド）の内容が右側（ディスティネーション・オペランド）へ転送されますが，コンピュータの転送命令はコピー命令であり，命令実行後，ソース側の内容は元の値を保持し，空（カラ）になるわけではありません．なおデータの移動方向は左から右であり，我々の習慣と同様です．

サイズはバイト，ワード，ロングワードを指定できますが，指定したサイズ以外のビットは変化せず，たとえばバイトを指定すれば，オペランドのビット7～0の1バイトが操作対象となり（上位のすべてのビットは操作の対象外となる），本命令によって影響を受けることはありません．

データ転送後，フラグは“N”と“Z”が変化し，その値がゼロであったとか，負であったとか，結果としてどのようなデータが転送されたのかを判別できます。

一方，“X”フラグは保持されるので，本命令に先行して実行された演算命令の桁上がり結果が（本命令によって）失われることがないように配慮されています．

転送先がアドレスレジスタである場合はMOVEAという専用命令があり，汎用のデータ転送とは区別されます．MOVEAはバイトオペランドがサポートされないこと，フラグ変化しない点などでMOVEと異なります（モトローラではアドレスと単なるデータとを区別するために専用のアドレス転送命令を用意したのでしょう）．

◉機械語フォーマット

15	14	13 12	11 10 9	8 7 6	5 4 3	2 1 0
0	0	サイズ	ディスティネーション実効アドレス		ソース実効アドレス	
			レジスタ	モード	モード	レジスタ

ソース実効アドレス（すべてのモード）

アドレッシングモード	対応ビット			
	5	4	3	2 1 0
Dn	0	0	0	レジスタ番号
An*	0	0	1	レジスタ番号
(An)	0	1	0	レジスタ番号
(An)+	0	1	1	レジスタ番号
−(An)	1	0	0	レジスタ番号
d16(An)	1	0	1	レジスタ番号
d8(An, IX)	1	1	0	レジスタ番号
AbS.W	1	1	1	0 0 0
AbS.L	1	1	1	0 0 1
d16(PC)	1	1	1	0 1 0
d8(PC, IX)	1	1	1	0 1 1
# Imm	1	1	1	1 0 0

*バイト・サイズ不可

ディスティネーション実効アドレス（データ可変モード）

アドレッシングモード	対応ビット			
	11 10 9	8	7	6
Dn	レジスタ番号	0	0	0
(An)	レジスタ番号	0	1	0
(An)+	レジスタ番号	0	1	1
−(An)	レジスタ番号	1	0	0
d16(An)	レジスタ番号	1	0	1
d8(An, IX)	レジスタ番号	1	1	0
Abs.W	0 0 0	1	1	1
Abs.L	0 0 1	1	1	1

サイズ	対応ビット	
	13	12
バイト	0	1
ワード	1	1
ロングワード	1	0

注）ディスティネーション実効アドレス部は，レジスタ，モードの順でマップされている．

CCR　X：変化せず
N：転送結果が負ならセット(1)，それ以外はリセット(0)
Z：転送結果がゼロならセット(1)，それ以外はリセット(0)
V：常にリセット(0)
C：常にリセット(0)

◉アドレッシング・モード

sou	size	dest Dn #	Dn ~	(An) #	(An) ~	(An)+ #	(An)+ ~	−(An) #	−(An) ~	d16(An) #	d16(An) ~	d8(An,Ix) #	d8(An,Ix) ~	Abs.W #	Abs.W ~	Abs.L #	Abs.L ~
Dn	B	2	4	2	8	2	8	2	8	4	12	4	14	4	12	6	16
	W	2	4	2	8	2	8	2	8	4	12	4	14	4	12	6	16
	L	2	4	2	12	2	12	2	12	4	16	4	18	4	16	6	20
An	B																
	W	2	4	2	8	2	8	2	8	4	12	4	14	4	12	6	16
	L	2	4	2	12	2	12	2	12	4	16	4	18	4	16	6	20
(An)	B	2	8	2	12	2	12	2	12	4	16	4	18	4	16	6	20
	W	2	8	2	12	2	12	2	12	4	16	4	18	4	16	6	20
	L	2	12	2	20	2	20	2	20	4	24	4	26	4	24	6	28
(An)+	B	2	8	2	12	2	12	2	12	4	16	4	18	4	16	6	20
	W	2	8	2	12	2	12	2	12	4	16	4	18	4	16	6	20
	L	2	12	2	20	2	20	2	20	4	24	4	26	4	24	6	28
−(An)	B	2	10	2	14	2	14	2	14	4	18	4	20	4	18	6	22
	W	2	10	2	14	2	14	2	14	4	18	4	20	4	18	6	22
	L	2	14	2	22	2	22	2	22	4	26	4	28	4	26	6	30
d16(An)	B	4	12	4	16	4	16	4	16	6	20	6	22	6	20	8	24
	W	4	12	4	16	4	16	4	16	6	20	6	22	6	20	8	24
	L	4	16	4	24	4	24	4	24	6	28	6	30	6	28	8	32
d8(An,IX)	B	4	14	4	18	4	18	4	18	6	22	6	24	6	22	8	26
	W	4	14	4	18	4	18	4	18	6	22	6	24	6	22	8	26
	L	4	18	4	26	4	26	4	26	6	30	6	32	6	30	8	34
Abs.W	B	4	12	4	16	4	16	4	16	6	20	6	22	6	20	8	24
	W	4	12	4	16	4	16	4	16	6	20	6	22	6	20	8	24
	L	4	16	4	24	4	24	4	24	6	28	6	30	6	28	8	32
Abs.L	B	6	16	6	20	4	20	6	20	8	24	8	26	8	24	10	28
	W	6	16	6	20	4	20	6	20	8	24	8	26	8	24	10	28
	L	6	20	6	28	4	28	6	28	8	32	8	34	8	32	10	36
d16(PC)	B	4	12	4	16	4	16	4	16	6	20	6	22	6	20	8	24
	W	4	12	4	16	4	16	4	16	6	20	6	22	6	20	8	24
	L	4	16	4	24	4	24	4	24	6	28	6	30	6	28	8	32
d8(PC,IX)	B	4	14	4	18	4	18	4	18	6	22	6	24	6	22	8	26
	W	4	14	4	18	4	18	4	18	6	22	6	24	6	22	8	26
	L	4	18	4	26	4	26	4	26	6	30	6	32	6	30	8	34
# Imm	B	4	8	4	12	4	12	4	12	6	16	6	18	6	16	8	20
	W	4	8	4	12	4	12	4	12	6	16	6	18	6	16	8	20
	L	6	12	6	20	6	20	6	20	8	24	8	26	8	24	10	28

◉サンプル・リスト

```
MOVE.L  D0,(A0)+
MOVE.B  D2,$4(A2)
MOVE.L  #$4888FFAA,$2000
```

2 ◉MOVEA [MOVE Address アドレスデータの転送]

MOVEA {.W／.L} <ea>, An

X	N	Z	V	C
—	—	—	—	—

解説

〈ea〉の内容を指定したアドレスレジスタへ転送しますが，ワード転送を指定した場合は32ビットに符号拡張され，指定されたアドレスレジスタへ転送されます．

ほとんどのアセンブラの文法ではMOVEの転送先にアドレスレジスタを指定すると，MOVEAと解釈しますが，“アドレス”という概念は大変重要であり，他のデータ転送とは区別した方がバグの発生を抑制できるので，

MOVE.L D0, A0

と記述できても，

MOVEA.L D0, A0

と記述した方がよいと（経験上）思われます．

CCR

X：変化せず
N：変化せず
Z：変化せず
V：変化せず
C：変化せず

◉アドレッシング・モード

sou	size	dest An #	dest An ～
Dn	B		
	W	2	4
	L	2	4
An	B		
	W	2	4
	L	2	4
(An)	B		
	W	2	8
	L	2	12
(An)+	B		
	W	2	8
	L	2	12
−(An)	B		
	W	2	10
	L	2	14
d16(An)	B		
	W	4	12
	L	4	16
d8(An,IX)	B		
	W	4	14
	L	4	18
Abs.W	B		
	W	4	12
	L	4	16
Abs.L	B		
	W	6	16
	L	4	20
d16(PC)	B		
	W	4	12
	L	4	16
d8(PC,IX)	B		
	W	4	14
	L	4	18
# Imm	B		
	W	4	8
	L	6	12

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	サイズ		ディスティネーション レジスタ			0	0	1	ソース実効アドレス モード			ソース実効アドレス レジスタ		

アドレス・レジスタ番号（000～111）

サイズ	対応ビット 13	対応ビット 12
ワード	1	1
ロングワード	1	0

ソース側（すべてのモード）

アドレッシングモード	5	4	3	2	1	0
Dn	0	0	0	レジスタ番号		
An	0	0	1	レジスタ番号		
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An, IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1
d16(PC)	1	1	1	0	1	0
d8(PC, IX)	1	1	1	0	1	1
# Imm	1	1	1	1	0	0

◉サンプル・リスト

```
MOVEA.L (AØ),A2
MOVEA.L #2ØØØ,A4
```

3 ◉MOVE to CCR [CCRへの転送]

MOVE {.W} <ea>, CCR	X	N	Z	V	C
	*	*	*	*	*

解説

〈ea〉で指定した内容をCCR（SR：ステータスレジスタの下位8ビット）へ転送するもので，通常は退避してあったフラグの復帰を行うのに使用されますが，5つのフラグを直接操作することも可能です．

転送サイズはワードですがCCRはビット4～0に意味があるため，ソース・オペランドの上位バイトはユーザ状態では意味を持ちませんし，本命令による影響も受けません．

CCR

X：ソース・オペランドの対応ビット（ビット4）の値が反映される

N：ソース・オペランドの対応ビット（ビット3）の値が反映される

Z：ソース・オペランドの対応ビット（ビット2）の値が反映される

V：ソース・オペランドの対応ビット（ビット1）の値が反映される

C：ソース・オペランドの対応ビット（ビット0）の値が反映される

◉アドレッシング・モード

sou	size	dest CCR #	dest CCR ～
Dn	B		
	W	2	12
	L		
(An)	B		
	W	2	16
	L		
(An)+	B		
	W	2	16
	L		
−(An)	B		
	W	2	18
	L		
d16(An)	B		
	W	4	20
	L		
d8(An,IX)	B		
	W	4	22
	L		
Abs.W	B		
	W	4	20
	L		
Abs.L	B		
	W	6	24
	L		
d16(PC)	B		
	W	4	20
	L		
d8(PC,IX)	B		
	W	4	22
	L		
# Imm	B		
	W	4	16
	L		

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	1	1	実効アドレス					
										モード			レジスタ		

実効アドレス（データ・モード）

アドレッシングモード	対応ビット 5	4	3	2	1	0
Dn	0	0	0	レジスタ番号		
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1
d16(PC)	1	1	1	0	1	0
d8(PC,IX)	1	1	1	0	1	1
# Imm	1	1	1	1	0	0

◉サンプル・リスト

```
MOVE     #0,CCR
```

4 ◉MOVE to SR［特権命令］［SRへの転送］

MOVE {.W} <ea>, SR	X	N	Z	V	C
	*	*	*	*	*

解説

〈ea〉で指定した内容をステータスレジスタ(SR)へ転送します。サイズはワードであり，ステータスレジスタのすべてのビットが影響を受けます。

68000の動作を決定する重要な命令であり，以下のステータス・ビットを操作することができますが，その意味に関しては必要な説明を参照してください。

SR

T：ソース・オペランドの対応ビット（ビット15，トレース）の値が反映される
S：ソース・オペランドの対応ビット（ビット13，スーパバイザ状態）の値が反映される
I_2：ソース・オペランドの対応ビット（ビット10，割り込みマスク）の値が反映される
I_1：ソース・オペランドの対応ビット（ビット9，割り込みマスク）の値が反映される
I_0：ソース・オペランドの対応ビット（ビット8，割り込みマスク）の値が反映される
X：ソース・オペランドの対応ビット(ビット4)の値が反映される
N：ソース・オペランドの対応ビット(ビット3)の値が反映される
Z：ソース・オペランドの対応ビット(ビット2)の値が反映される
V：ソース・オペランドの対応ビット(ビット1)の値が反映される
C：ソース・オペランドの対応ビット(ビット0)の値が反映される

◉アドレッシング・モード

sou	size	dest SR #	dest SR ~
Dn	B		
	W	2	12
	L		
(An)	B		
	W	2	16
	L		
(An)+	B		
	W	2	16
	L		
−(An)	B		
	W	2	18
	L		
d16(An)	B		
	W	4	20
	L		
d8(An,IX)	B		
	W	4	22
	L		
Abs.W	B		
	W	4	20
	L		
Abs.L	B		
	W	6	24
	L		
d16(PC)	B		
	W	4	20
	L		
d8(PC,IX)	B		
	W	4	22
	L		
#Imm	B		
	W	4	16
	L		

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	0	1	1	実効アドレス					
										モード			レジスタ		

実効アドレス(データ・モード)

アドレッシングモード	対応ビット 5	4	3	2	1	0
Dn	0	0	0	レジスタ番号		
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1
d16(PC)	1	1	1	0	1	0
d8(PC,IX)	1	1	1	0	1	1
# Imm	1	1	1	1	0	0

◉サンプル・リスト

```
MOVE    (A0),SR
MOVE    $1000,SR
```

5 ◉MOVE from SR [SRからの転送]

MOVE {.W} SR ,<ea>	X	N	Z	V	C
	—	—	—	—	—

解説　ステータスレジスタ（SR）の内容を〈ea〉で指定した場所へ転送（退避）します．オペランドサイズはワードであり，フラグは変化しません．

68000では特権命令ではありませんが，68010では特権命令となっています．

SR
T：変化せず
S：変化せず
I_2：変化せず
I_1：変化せず
I_0：変化せず

X：変化せず
N：変化せず
Z：変化せず
V：変化せず
C：変化せず

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	1	1	実効アドレス					
										モード			レジスタ		

実効アドレス（データ・可変モード）

アドレッシングモード	対応ビット 5	4	3	2	1	0
Dn	0	0	0	レジスタ番号		
(An)	0	1	0	レジスタ番号		
(An)＋	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1

◉アドレッシング・モード

sou	size	dest Dn #	Dn ~	(An) #	(An) ~	(An)+ #	(An)+ ~	−(An) #	−(An) ~	d16(An) #	d16(An) ~	d8(An,IX) #	d8(An,IX) ~	Abs.W #	Abs.W ~	Abs.L #	Abs.L ~
SR	B																
	W	2	6	2	12	2	12	2	14	4	16	4	18	4	16	6	20
	L																

◉サンプル・リスト

```
MOVE     SR,DØ
MOVE     SR,$1ØØØ
```

6 ◉MOVE from USP［特権命令］［USPからの転送］

MOVE ｛.L｝ USP, An	X	N	Z	V	C
	—	—	—	—	—

解 説　ユーザ・スタック・ポインタ(USP, A7)の内容を指定したアドレスレジスタへ転送します．USPは32ビットですから扱うサイズはロングワードです．

◉アドレッシング・モード

sou	size	dest An #	dest An ～
USP	B		
	W		
	L	2	4

CCR
X：変化せず
N：変化せず
Z：変化せず
V：変化せず
C：変化せず

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	0	dr	レジスタ		

レジスタ → アドレス・レジスタ番号（０００～１１１）
dr → １：(USP→An)

◉サンプル・リスト

```
MOVE     USP,A5
```

APPENDIX ◉MOVE from CCR

MOVE ｛.W｝ CCR, 〈ea〉	X	N	Z	V	C
	—	—	—	—	—

解 説　CCRの内容を〈ea〉で指定した場所へ転送しますが，オペレーションはワードであり，上位バイトは$00，下位バイトがCCRの内容となります．

●この命令は，68000ではサポートされません．

CCR
X：変化せず
N：変化せず
Z：変化せず
V：変化せず
C：変化せず

7 ◉MOVE to USP［特権命令］［USPへの転送］

MOVE ｛.L｝ An, USP	X	N	Z	V	C
	—	—	—	—	—

解説　指定したアドレスレジスタの内容をユーザ・スタック・ポインタ（USP, A7）へ転送（設定）します．USPは32ビットですから扱うサイズはロングワードです．

CCR
X：変化せず
N：変化せず
Z：変化せず
V：変化せず
C：変化せず

◉アドレッシング・モード

sou	size	dest USP #	dest USP ～
An	B		
	W		
	L	2	4

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	0	dr	レジスタ		

レジスタ → アドレスレジスタ番号（０００～１１１）
dr → 0：(An→USP)

◉サンプル・リスト

```
MOVE    AØ,USP
```

8 ◉MOVEM from reg [Move Multiple registers 複数レジスタの同時転送]

MOVEM {.W／.L} 〈レジスタ・リスト〉,〈ea〉	X	N	Z	V	C
	—	—	—	—	—

解説

本命令は複数のレジスタ群をメモリ領域へ退避する命令です．

▶制御モードに含まれるアドレス形式のいずれかを用いた場合，転送は指定したメモリロケーションから始まって，アドレスの増加する方向にレジスタ群の内容を転送しますが，転送順は D0～D7，A0～A7の順序であり，このうちの指定されたレジスタ(マスクフィールドが有効）だけが対象になります．

▶プリデクリメント・アドレスレジスタ間接モード“－(An)”の場合は，“PUSH”ですから，「レジスタ群→メモリ」への一方通行となり，サブルーチンの入り口で必要なレジスタを退避する場合に便利です．

この場合は，指定したメモリロケーション－2（ワードサイズ）または－4（ロングワードサイズ）から始まり，アドレスの減少する方向に，アドレスレジスタA7～A0，データレジスタD7～D0の順で転送されます．デクリメントされたアドレスレジスタは更新され，最後に格納したレジスタのアドレスをポイントします．

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	dr	0	0	1	Sz	実効アドレス					
										モード			レジスタ		

dr → 0：(レジスタ→メモリ)

Sz → 0：ワード転送
1：ロングワード転送

実効アドレス(制御・可変またはプリデクリメント・モード)

アドレッシングモード	対応ビット 5	4	3	2	1	0
(An)	0	1	0	レジスタ番号		
－(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1

〈第２ワード：レジスタ・リスト・マスク・フィールド〉

オブジェクト・コードの次のワードはレジスタ・マスク・フィールドが続き、１ビットがレジスタ１個を表し、転送順位は、下位ビットに対応するレジスタから割り当てられます（対応するレジスタ番号ビットは“1”）．
マスクフィールドの対応するヒットかセット（1）されているレジスタが転送対象であり，さらにこのフィールドは転送順も管理します．すなわち，下位ビットから最初に転送されるレジスタがマップされ，上位ビットが最後に転送されるレジスタに対応します．

●制御・可変モードのレジスタ・リスト・マスク・フィールド

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A7	A6	A5	A4	A3	A2	A1	A0	D7	D6	D5	D4	D3	D2	D1	D0

●プリデクリメント・モードのレジスタ・リスト・マスク・フィールド

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D0	D1	D2	D3	D4	D5	D6	D7	A0	A1	A2	A3	A4	A5	A6	A7

CCR

X：変化せず
N：変化せず
Z：変化せず
V：変化せず
C：変化せず

◉アドレッシング・モード

sou	size	dest											
		(An)		−(An)		d16(An)		d8(An,IX)		Abs.W		Abs.L	
		#	~	#	~	#	~	#	~	#	~	#	~
Dn	B												
	W	4	8+4n	4	8+4n	6	12+4n	6	14+4n	6	12+4n	8	16+4n
	L	4	8+8n	4	8+8n	6	12+8n	6	14+8n	6	12+8n	8	16+8n
An	B												
	W	4	8+4n	4	8+4n	6	12+4n	6	14+4n	6	12+4n	8	16+4n
	L	4	8+8n	4	8+8n	6	12+8n	6	14+8n	6	12+8n	8	16+8n

◉サンプル・リスト

```
MOVEM.L D0-D7/A0-A7,-(SP)
```

9 ◉MOVEM to reg [Move Multiple registers 複数レジスタとの同時転送]

MOVEM {.W/.L} <ea>,<レジスタ・リスト>	X	N	Z	V	C
	—	—	—	—	—

解説

メモリ領域に退避しておいたレジスタ群を復帰する命令です．

▶ワードサイズの場合，アドレスレジスタもデータレジスタも，復帰される値は符号拡張され，結果的に32ビットの値がレジスタへロード（転送）されます．

したがって，上位ワードは$0000または$FFFFで満たされることになることから，レジスタ群のメモリへの退避/復帰はロングワードで行い，ワードサイズを指定しない方が賢明でしょう．

▶制御モードに含まれるアドレス形式のいずれかを用いた場合，転送は指定したメモリロケーションから始まって，アドレスの増加する方向にレジスタ群の内容を転送しますが，転送順はD0～D7，A0～A7の順序であり，このうちの指定されたレジスタ(マスクフィールドが有効)だけが対象になります．

▶ポストインクリメント・アドレッシング“(An)+”は“POP”であり，「メモリ──→レジスタ群」への一方通行です．

この場合は，指定したメモリロケーションから始まって，アドレスの増加する方向に向かってD0～D7，A0～A7の順で転送します．それゆえ，アドレッシングに使用したアドレスレジスタは，命令の実行完了時には最後に転送したアドレス＋2（ワードサイズ）または＋4（ロングワードサイズ）をポイントします．

◉アドレッシング・モード

sou	size	dest			
		Dn		An	
		#	～	#	～
(An)	B				
	W	4	12+4n	4	12+4n
	L	4	12+8n	4	12+8n
(An)+	B				
	W	4	12+4n	4	12+4n
	L	4	12+8n	4	12+8n
d16(An)	B				
	W	6	16+4n	6	16+4n
	L	6	16+8n	6	16+8n
d8(An,IX)	B				
	W	6	18+4n	6	18+4n
	L	6	18+8n	6	18+8n
Abs.W	B				
	W	6	16+4n	6	16+4n
	L	6	16+8n	6	16+8n
Abs.L	B				
	W	8	20+4n	8	20+4n
	L	8	20+8n	8	20+8n
d16(PC)	B				
	W	6	16+4n	6	16+4n
	L	6	16+8n	6	16+8n
d8(PC,IX)	B				
	W	6	18+4n	6	18+4n
	L	6	18+8n	6	18+8n

▶ MOVE <レジスタ・リスト>,<ea>

と

MOVE <ea>,<レジスタ・リスト>

とはペアで使用するのが前提ですから，このことを理解すれば，どのような転送モードを選択すべきかを理解できるでしょう．たとえばサブルーチンの入り口では“−(An)”でレジスタ群をメモリ領域へ退避し，出口では必然的に“(An)+”で復帰する，というのが通常の選択です．

CCR

X：変化せず
N：変化せず
Z：変化せず
V：変化せず
C：変化せず

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	dr	0	0	1	Sz	実効アドレス モード			レジスタ		

Sz → 0：ワード転送 1：ロングワード転送

dr → 1：(メモリ→レジスタ)

実効アドレス(制御またはポストインクリメント・モード)

アドレッシングモード	対応ビット					
	5	4	3	2	1	0
(An)	0	1	0	レジスタ番号		
(An)＋	0	1	1	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
AbS.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1
d16(PC)	1	1	1	0	1	0
d8(PC,IX)	1	1	1	0	1	1

〈第２ワード：レジスタ・リスト・マスク・フィールド〉

オブジェクト・コードの次のワードは，レジスタ・リスト・マスク・フィールドが続きます．１ビットがレジスタ１個を表し，転送順は下位ビットに対応するレジスタから割り当てられます.(対応するビット番号は"1").

マスクフィールドの対応するビットがセット（１）されているレジスタが転送対象であり，さらにこのフィールドは転送順も管理します．すなわち，下位ビットから最初に転送されるレジスタがマップされ，上位ビットが最後に転送されるレジスタに対応します．

●制御またはポストインクリメント・モードのレジスタ・リスト・マスク・フィールド

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A7	A6	A5	A4	A3	A2	A1	A0	D7	D6	D5	D4	D3	D2	D1	D0

◉サンプル・リスト

```
MOVEM.L (SP)+,DØ-D7/AØ-A7
```

10 ●MOVEP to Dn ［MOVE Peripheral data 周辺データの転送］

MOVEP {.W／.L} d16 (An), Dn	X	N	Z	V	C
	—	—	—	—	—

解説

“d16(An)”で指定されるメモリロケーションから始まり，2つずつ増加するような1番地おきのバイト配列との間でデータ転送するものですが，このフィールドで指定するアドレスは，多くの場合I/Oアドレスであることを想定した命令です．

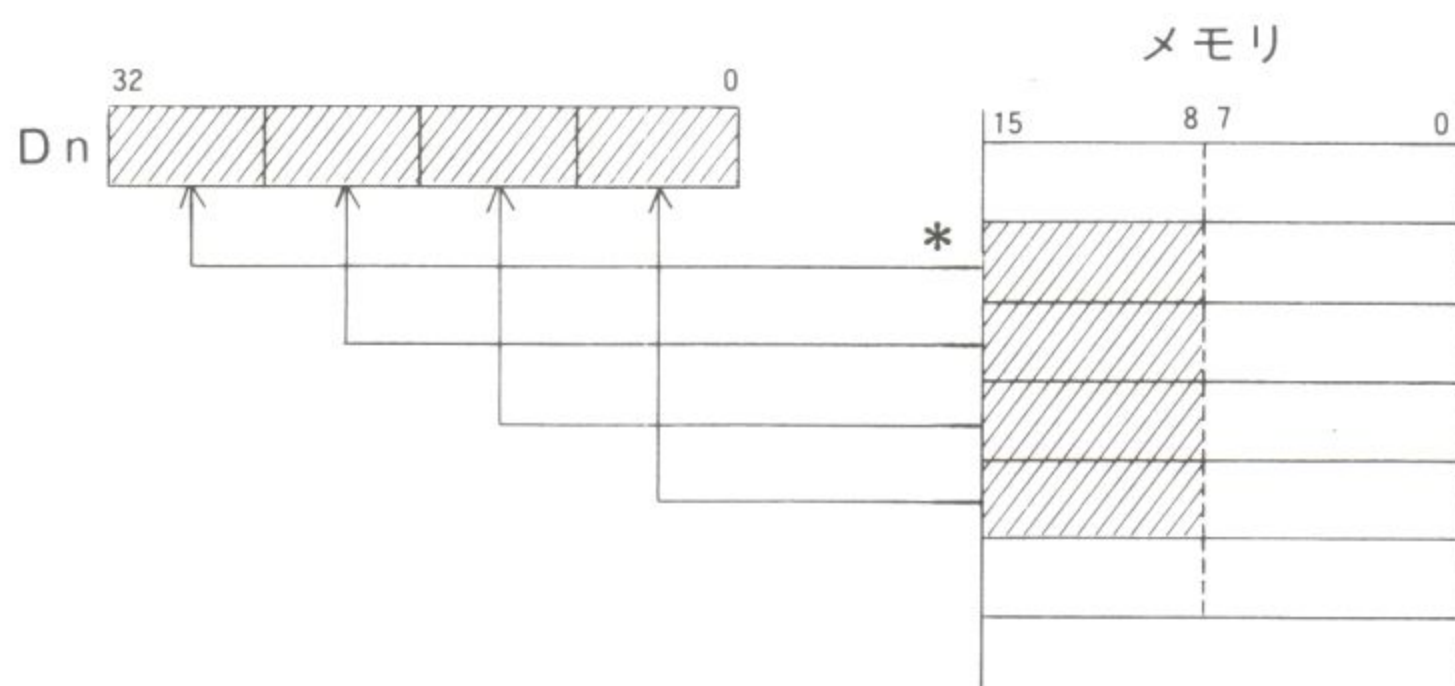
I/Oディバイスとの入/出力は本質的にバイトの属性があり，このため，上位バイトまたは下位バイトに対応するロケーションだけが使用される点で，I/Oディバイスへの書き込みに便利な命令となっています．

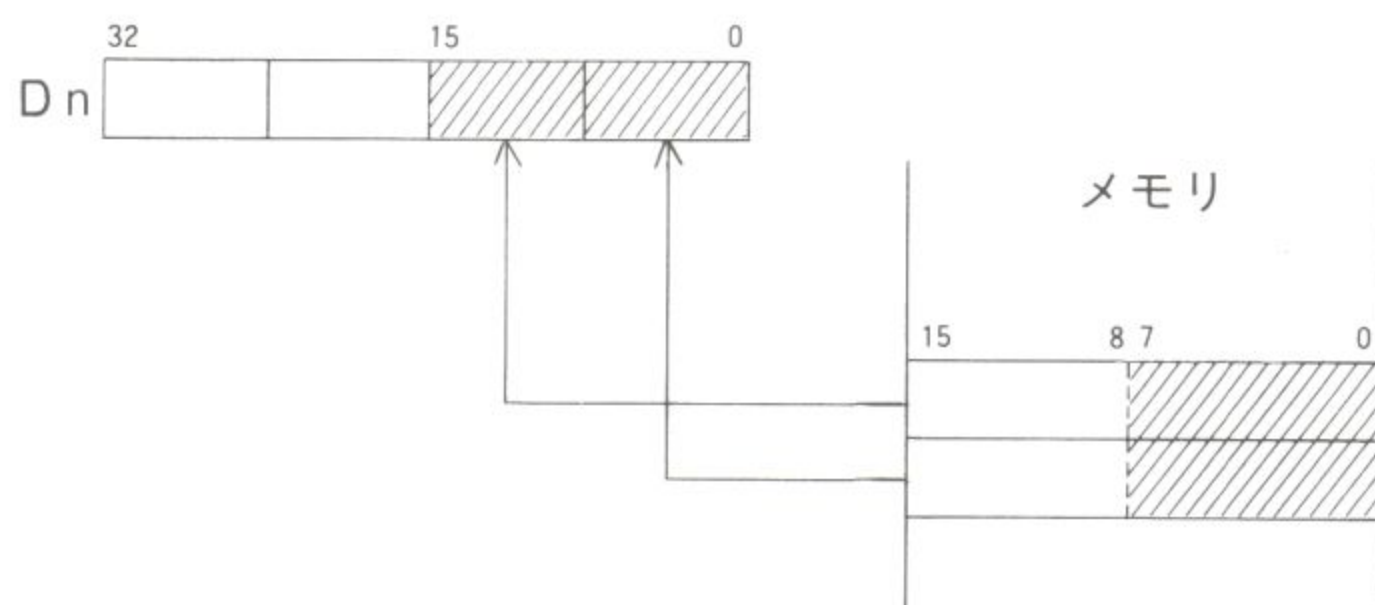
▶データレジスタへは2バイト（ワード）または4バイト（ロングワード）が読み込まれるわけですが，ゼロ番地に近い方がデータレジスタの上位に対応します．

▶“d16(An)”に設定されるロケーションが偶数であれば，そこから偶数アドレスの番地だけをアドレスの増加する方向へ2バイトまたは4バイト読み出し，指定されたデータレジスタへ転送します．したがって，下位データバスは使用されず，奇数アドレスはスキップされます．

▶“d16(An)”へ設定されるロケーションが奇数であれば，そこから奇数アドレスの番地だけをアドレスの増加する方向へ2バイトまたは4バイト読み出し，指定されたデータレジスタへ転送します．したがって，上位データバスは使用されず，偶数アドレスはスキップされます．

〈実行の様子〉

例1：偶数ロケーションとデータ・レジスタ（サイズはロングワード）

＊のアドレスは d16(An)で指定される偶数アドレスを意味する．
奇数番地（データ・バスのビット7～0）は使用されずにスキップされる．

例2：奇数ロケーションとデータ・レジスタ（サイズはワード）

＊＊のアドレスは d16(An)で指定される奇数アドレスを意味する．
偶数番地（データ・バスのビット15～8）は使用されずにスキップされる．

CCR X：変化せず
N：変化せず
Z：変化せず
V：変化せず
C：変化せず

◉アドレッシング・モード

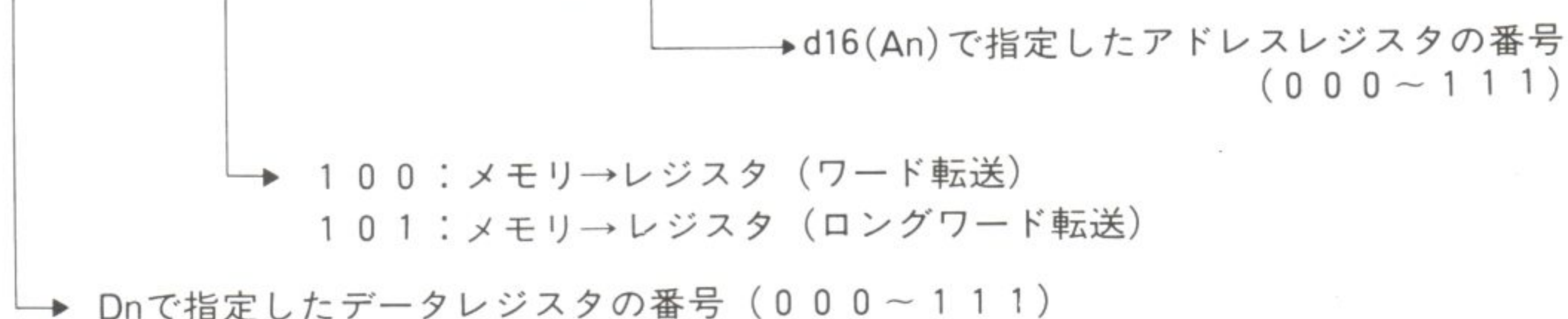
sou	size	dest	
		Dn	
		♯	〜
d16(An)	B		
	W	4	16
	L	4	24

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	データレジスタ			OPモード			0	0	1	アドレスレジスタ		

アドレスレジスタ → d16(An)で指定したアドレスレジスタの番号（０００〜１１１）

OPモード → １００：メモリ→レジスタ（ワード転送）
１０１：メモリ→レジスタ（ロングワード転送）

データレジスタ → Dnで指定したデータレジスタの番号（０００〜１１１）

〈第２ワード：ディスプレースメント・フィールド〉

オブジェクト・コードの第２ワードにはオペランドのロケーションを計算する際に使用されるディスプレースメントが格納されます．

◉サンプル・リスト

```
MOVEP.W $18(A5),DØ
```

第3部 命令の詳細

11●MOVEP from Dn [Move Peripheral data 周辺データの転送]

MOVEP {.W／.L} Dn, d16（An）	X	N	Z	V	C
	—	—	—	—	—

解説　データレジスタと指定したメモリロケーションから始まり，２つずつ増加するような１番地おきのバイト配列との間でデータ転送するものですが，d16（An）で指定するアドレスは，多くの場合I/Oアドレスであることを想定した命令です．

I/Oディバイスとの入/出力は本質的にバイトの属性があり，このため，上位バイトまたは下位バイトに対応するロケーションだけが使用される点で，I/Oディバイスへの書き込みに便利な命令となっています．

▶データレジスタのサイズはワードとロングワードであり，２バイトまたは４バイトがメモリ（I/O）のどこかへ転送されますが，データレジスタの上位バイトはゼロ番地に近いアドレスへ，下位バイトはゼロ番地から遠いアドレスへ，それぞれ転送されます．

▶ロケーションが偶数であれば偶数アドレスだけが使用され，データバスの上位だけが使用されます．奇数であれば奇数アドレスだけが使用され，データバスの下位半分だけで転送が行われます．

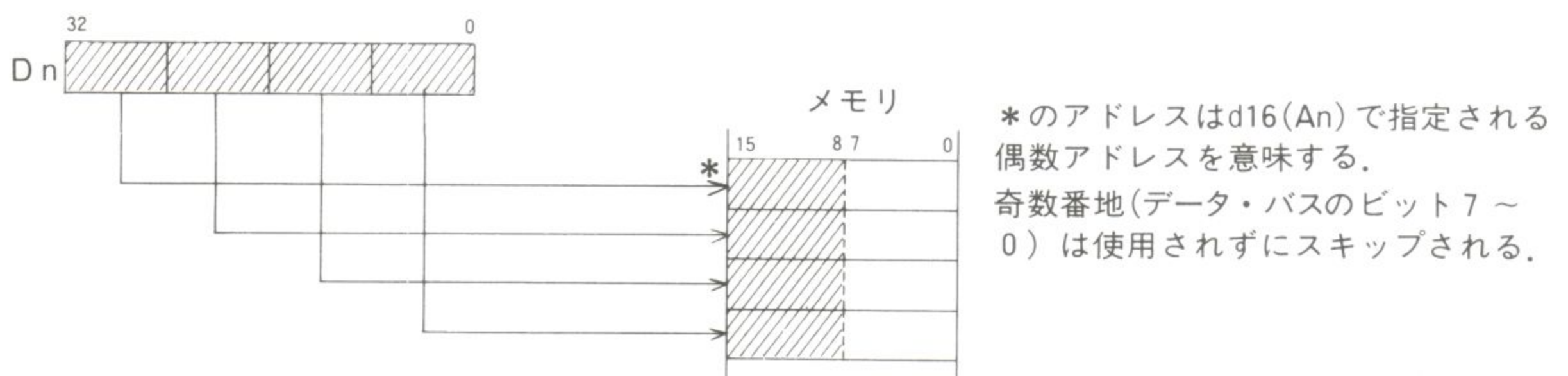
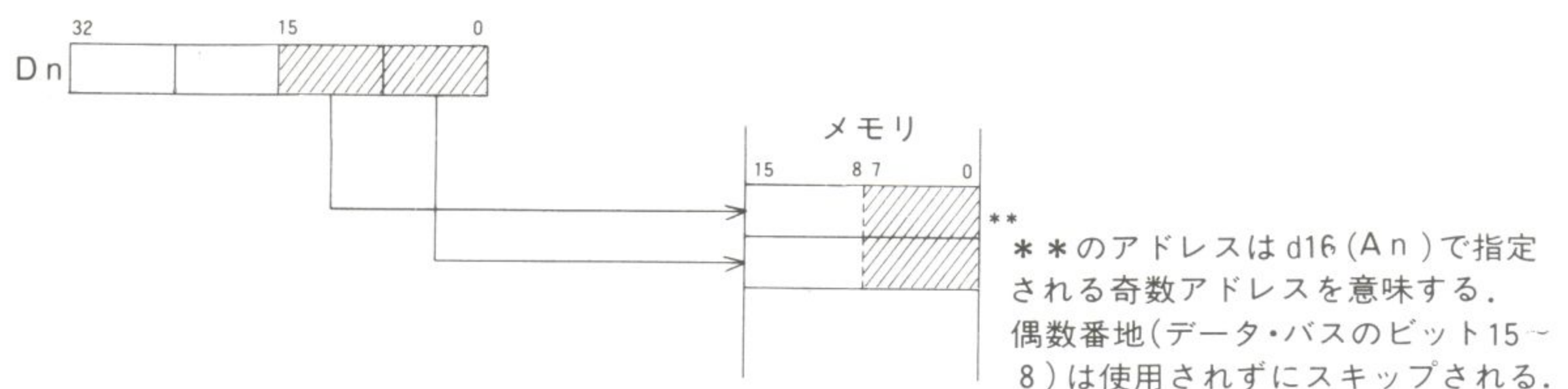
<実行の様子>

例１：偶数ロケーションとデータレジスタ（サイズはロングワード）

＊のアドレスはd16(An)で指定される偶数アドレスを意味する．
奇数番地（データ・バスのビット７～０）は使用されずにスキップされる．

例２：奇数ロケーションとデータレジスタ（サイズはワード）

＊＊のアドレスはd16（An）で指定される奇数アドレスを意味する．
偶数番地（データ・バスのビット15～８）は使用されずにスキップされる．

CCR
X：変化せず
N：変化せず
Z：変化せず
V：変化せず
C：変化せず

●アドレッシング・モード

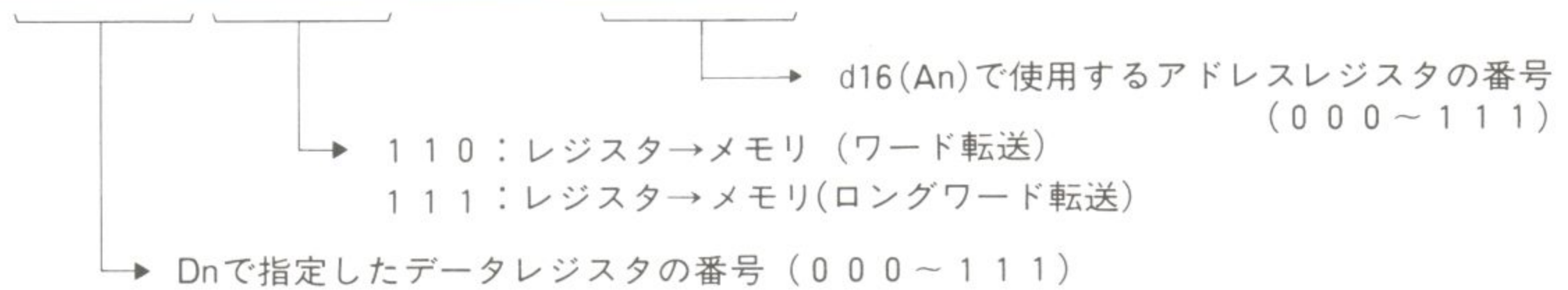
sou	size	dest	
		d16(An)	
		#	～
Dn	B		
	W	4	16
	L	4	24

●機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	データレジスタ			OPモード			0	0	1	アドレスレジスタ		

アドレスレジスタ → d16(An)で使用するアドレスレジスタの番号（０００～１１１）

OPモード → １１０：レジスタ→メモリ（ワード転送）
１１１：レジスタ→メモリ(ロングワード転送)

データレジスタ → Dnで指定したデータレジスタの番号（０００～１１１）

〈第２ワード：ディスプレースメント・フィールド〉

オブジェクト・コードの第２ワードには，オペランドのロケーションを計算する際に使用するディスプレースメントが格納されます．

●サンプル・リスト

```
MOVEP.W DØ,$18(A5)
```

第3部 命令の詳細

12 ◉MOVEQ [Move Quick クイック転送]

MOVEQ {.L} ＃ <data>, Dn

X	N	Z	V	C
—	*	*	0	0

解説

1バイトのイミディエイトデータ（即値）をデータレジスタへ転送する命令ですが，データレジスタは32ビットが使用され，指定した1バイトのデータは32ビットに符号拡張されてデータレジスタへ転送されます．

通常の転送命令よりはオブジェクトコードが短く高速に実行できる命令です．

CCR

X：変化せず
N：転送結果が負ならセット(1)，それ以外はリセット(0)
Z：転送結果がゼロならセット(1)，それ以外はリセット(0)
V：常にリセット(0)
C：常にリセット(0)

◉アドレッシング・モード

sou	size	dest Dn ＃	dest Dn ～
＃Imm	B		
	W		
	L	2	4

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	レジスタ			0	データ							

データ → ＃<data>で指定した１バイトのイミディエイト値

レジスタ → Dnで指定したデータレジスタ番号（０００～１１１）

◉サンプル・リスト

```
MOVEQ    #$4A,D0
```

13 ◉EXG [Exchang registers　レジスタ交換]

EXG {.L} <reg>, <reg>

X	N	Z	V	C
—	—	—	—	—

解説

２つのレジスタの内容を交換しますが，32ビット全部が操作されます．
次の３通りの方法を指定できます．
●データレジスタ同士の交換．
●アドレスレジスタ同士の交換．
●データレジスタとアドレスレジスタとの交換．

CCR

X：変化せず
N：変化せず
Z：変化せず
V：変化せず
C：変化せず

◉アドレッシング・モード

sou	size	dest Dn #	dest Dn ～	dest An #	dest An ～
Dn	B				
	W				
	L	2	6	2	6
An	B				
	W				
	L	2	6	2	6

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	ソースレジスタ			1	OPモード					ディスティネーションレジスタ		

ディスティネーション・オペランド（第２オペランド）で指定したデータレジスタあるいはアドレスレジスタの番号（０００～１１　）．
データレジスタとアドレスレジスタが交換される場合は，常にアドレスレジスタの番号（０００～１１１）がマップされる．

交換の内容	対応ビット 7	6	5	4	3
データレジスタ同士	0	1	0	0	0
アドレスレジスタ同士	0	1	0	0	1
データレジスタとアドレスレジスタ	1	0	0	0	1

ソースオペランド（第１オペランド）で指定したデータレジスタあるいはアドレスレジスタ番号（０００～１１１）．
データレジスタとアドレスレジスタを交換する場合は，常にデータレジスタの番号（０００～１１１）がマップされる．

◉サンプル・リスト

```
EXG     DØ,D1
```

第3部 命令の詳細

14 ◉SWAP [Swap register halves レジスタ半分交換]

SWAP {.W} Dn

X	N	Z	V	C
—	＊	＊	0	0

解説 指定したデータレジスタの上位ワードと下位ワードを交換します.

CCR
X：変化せず
N：演算の結果，データの最上位ビット（MSB）が〝1〞ならセット(1)，それ以外はリセット(0)
Z：演算の結果がゼロならセット(1)，それ以外はリセット(0)
V：常にリセット(0)
C：常にリセット(0)

◉アドレッシング・モード

sou	size	dest Dn #	dest Dn ～
	B		
	W	2	4
	L		

＊ソースオペランドは存在しない.

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	1	0	0	0	レジスタ		

→ Dnで指定したデータレジスタ番号（000～111）

◉サンプル・リスト

```
SWAP    D0
```

15 ●LEA [Load Effective Address 実効アドレスのロード]

LEA {.L} 〈ea〉, An

X	N	Z	V	C
—	—	—	—	—

解説

〈ea〉で指定した実効アドレスをAnへ転送します.

実効アドレスはロングワードであり, Anの32ビットすべてが影響を受けます.

CCR

X：変化せず
N：変化せず
Z：変化せず
V：変化せず
C：変化せず

◉アドレッシング・モード

sou	size	dest An #	dest An ~
(An)	B		
	W		
	L	2	4
d16(An)	B		
	W		
	L	4	8
d8(An,IX)	B		
	W		
	L	4	12
Abs.W	B		
	W		
	L	4	8
Abs.L	B		
	W		
	L	6	12
d16(PC)	B		
	W		
	L	4	8
d8(PC,IX)	B		
	W		
	L	4	12

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	レジスタ			1	1	1	実効アドレス モード			実効アドレス レジスタ		

Anで指定したアドレスレジスタの番号（000～111）

実効アドレス（制御モード）

アドレッシングモード	対応ビット 5	4	3	2	1	0
(An)	0	1	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1
d16(PC)	1	1	1	0	1	0
d8(PC,IX)	1	1	1	0	1	1

◉サンプル・リスト

```
LEA     2Ø(A6),AØ
LEA     (A2),A5
LEA     LABEL,A3
```

LABELは記号番地を意味する

16◉PEA [Push Effective Address 実効アドレスのスタック]

PEA {.L} <ea>	X	N	Z	V	C
	—	—	—	—	—

解説 〈ea〉で指定した実効アドレスをシステムスタックへプッシュ（転送）します。

システム・スタック・ポインタ(SP)が暗黙のうちに使用されます。

CCR

X：変化せず
N：変化せず
Z：変化せず
V：変化せず
C：変化せず

◉アドレッシング・モード

sou	size	dest #	dest ~
(An)	B		
	W		
	L	2	14
d16(An)	B		
	W		
	L	4	18
d8(An,IX)	B		
	W		
	L	4	22
Abs.W	B		
	W		
	L	4	18
Abs.L	B		
	W		
	L	6	22
d16(PC)	B		
	W		
	L	4	18
d8(PC,IX)	B		
	W		
	L	4	22

destは存在しない

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	1	実効アドレス					
										モード			レジスタ		

実効アドレス（制御モード）

アドレッシングモード	対応ビット 5	4	3	2	1	0
(An)	0	1	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1
d16(PC)	1	1	1	0	1	0
d8(PC,IX)	1	1	1	0	1	1

◉サンプル・リスト

```
PEA     2Ø(A6)
PEA     LABEL
```

LABELは記号番地を意味する

17 ◉LINK [Link and allocate　スタック・リンクと割付け]

LINK An, ＃〈エリア長〉	X	N	Z	V	C
	—	—	—	—	—

解 説

ネストされたサブルーチン用にローカルエリアを確保します．

メインからの引数をサブルーチンへ渡し，サブルーチン内ではメイン側へ影響を与えることなく作業エリアを確保できるなど，きわめて有用な命令です．

〈エリア長〉の値はシステムスタックを破壊しないためにマイナスを指定する必要があり，−32768～0の範囲を指定することになります．

次のような一連の処理を行います．

① Anで指定したアドレスレジスタをシステムスタックへプッシュする（この時点でSPは4だけ減少している）．

② 更新されたSPの内容をAnへ転送する．

③ SPに〈エリア長〉を加算する．

CCR

X：変化せず
N：変化せず
Z：変化せず
V：変化せず
C：変化せず

◉アドレッシング・モード

sou	size	dest #〈エリア長〉 #	dest #〈エリア長〉 ～
An			
		4	16

sizeは存在しない

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	0	1	0	レジスタ		

→ Anで指定したアドレスレジスタ番号（000～111）

〈第2ワード：エリア長〉

エリア長フィールドで指定した値（2バイトの符号付整数）は，
オブジェクトコードの第2ワードに格納されます．

15 … 0
エ　リ　ア　長

◉サンプル・リスト

```
LINK    A6,#-8Ø
```

18 ◉UNLK [Unlink リンク解除]

UNLK An

X	N	Z	V	C
—	—	—	—	—

解説

LINK命令で確保したスタックエリアを開放するものです．
次のような一連の処理を行います．
① 指定したアドレスレジスタの内容をスタックポインタへ転送（復帰）する．
② スタックから取り出した4バイトを指定したアドレスレジスタへ転送（復帰）する．

CCR

X：変化せず
N：変化せず
Z：変化せず
V：変化せず
C：変化せず

◉アドレッシング・モード

sou	size	dest	
		#	～
An			
		2	12

sizeに無関係

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	0	1	1	レジスタ		

レジスタ → リンク解除（アンリンク）を行う時に使用するアドレスレジスタ番号で、Anで指定したものです(０００～１１１)．

◉サンプル・リスト

```
UNLK    A6
```

19 ◉ADD [Add binary 2進加算]

ADD {.B／.W／.L} Dn, <ea>	X	N	Z	V	C
	*	*	*	*	*

解説 ディスティネーション・オペランドとソース・オペランドを加算し，結果をディスティネーションへ格納します．

CCR
X：Cビットと同じ値
N：演算結果が負ならセット(1)，それ以外はリセット(0)
Z：演算結果がゼロならセット(1)，それ以外はリセット(0)
V：オーバフローが発生すればセット(1)，それ以外はリセット(0)
C：桁上がり（キャリ）が発生すればセット(1)，それ以外はリセット(0)

◉機械語フォーマット

15	14	13	12	11 10 9	8 7 6	5 4 3	2 1 0
1	1	0	1	レジスタ	OPモード	実効アドレス モード	実効アドレス レジスタ

レジスタ → 指定したデータレジスタ番号 000～111

サイズ	対応ビット 8	7	6
バイト	1	0	0
ワード	1	0	1
ロングワード	1	1	0

ディスティネーション実効アドレス(メモリ可変モード)

アドレッシングモード	対応ビット 5	4	3	2	1	0
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1

◉アドレッシング・モード

sou	size	dest (An) #	～	(An)+ #	～	−(An) #	～	d16(An) #	～	d8(An,IX) #	～	Abs.W #	～	Abs.L #	～
Dn	B	2	12	2	12	2	14	4	16	4	18	4	16	6	20
	W	2	12	2	12	2	14	4	16	4	18	4	16	6	20
	L	2	20	2	20	2	22	4	24	4	26	4	24	6	28

◉サンプル・リスト

```
ADD      DØ,(AØ)
```

20◉ADD [Add binary 2進加算]

ADD {.B／.W／.L} 〈ea〉, Dn

X	N	Z	V	C
*	*	*	*	*

解 説

ディスティネーション・オペランドとソース・オペランドを加算し，結果をディスティネーションへ格納します．

ソースオペランドの〈ea〉にAn（アドレスレジスタ直接形式）を指定した場合，オペレーションサイズはワードまたはロングワードであり，バイトサイズは許されません．

CCR

X：Cビットと同じ値
N：演算結果が負ならセット(1)，それ以外はリセット(0)
Z：演算結果がゼロならセット(1)，それ以外はリセット(0)
V：オーバフローが発生すればセット(1)，それ以外はリセット(0)
C：桁上がり（キャリ）が発生すればセット(1)，それ以外はリセット(0)

◉アドレッシング・モード

sou	size	dest Dn #	dest Dn ～
Dn	B	2	4
	W	2	4
	L	2	8
An	B		
	W	2	4
	L	2	8
(An)	B	2	8
	W	2	8
	L	2	14
(An)+	B	2	8
	W	2	8
	L	2	14
−(An)	B	2	10
	W	2	10
	L	2	16
d16(An)	B	4	12
	W	4	12
	L	4	18
d8(An,IX)	B	4	14
	W	4	14
	L	4	20
Abs.W	B	4	12
	W	4	12
	L	4	18
Abs.L	B	6	16
	W	6	16
	L	6	22
d16(PC)	B	4	12
	W	4	12
	L	4	18
d8(PC,IX)	B	4	14
	W	4	14
	L	4	20
#Imm (注)	B	4	8
	W	4	8
	L	6	14

◉機械語フォーマット

15	14	13	12	11 10 9	8 7 6	5 4 3	2 1 0
1	1	0	1	レジスタ	OPモード	実効アドレス モード	実効アドレス レジスタ

レジスタ → 指定したデータレジスタ番号

サイズ	対応ビット 8	7	6
バイト	0	0	0
ワード	0	0	1
ロングワード	0	1	0

ソース実効アドレス（すべてのモード）

アドレッシングモード	対応ビット 5	4	3	2	1	0
Dn	0	0	0	レジスタ番号		
An*	0	0	1	レジスタ番号		
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1
d16(PC)	1	1	1	0	1	0
d8(PC,IX)	1	1	1	0	1	1
(注)# Imm	1	1	1	1	0	0

＊ バイトサイズ不可
（注）ADDIを使用する

◉サンプル・リスト

```
ADD     (A2)+,D0
```

21◉ADDA [Add Address アドレス・データの加算]

ADDA {.W/.L} <ea>, An	X	N	Z	V	C
	—	—	—	—	—

解説　Anとソース・オペランドを加算し，結果をディスティネーションへ格納します．

サイズがワードの場合，〈ea〉の内容は32ビットに符号拡張され，アドレスレジスタの全32ビットが演算の対象になります．

CCR
X：変化せず
N：変化せず
Z：変化せず
V：変化せず
C：変化せず

◉アドレッシング・モード

sou	size	dest An #	dest An ~
Dn	B		
	W	2	8
	L	2	8
An	B		
	W	2	8
	L	2	8
(An)	B		
	W	2	12
	L	2	14
(An)+	B		
	W	2	12
	L	2	14
−(An)	B		
	W	2	14
	L	2	16
d16(An)	B		
	W	4	16
	L	4	18
d8(An,IX)	B		
	W	4	18
	L	4	20
Abs.W	B		
	W	4	16
	L	4	18
Abs.L	B		
	W	6	20
	L	6	22
d16(PC)	B		
	W	4	16
	L	4	18
d8(PC,IX)	B		
	W	4	18
	L	4	20
#Imm	B		
	W	4	12
	L	6	14

◉機械語フォーマット

15	14	13	12	11 10 9	8 7 6	5 4 3	2 1 0
						実効アドレス	
1	1	0	1	レジスタ	OPモード	モード	レジスタ

レジスタ → 指定したアドレスレジスタの番号（000〜111）

OPモード →

サイズ	対応ビット 8	7	6
ワード	0	1	1
ロングワード	1	1	1

実効アドレス（すべてのモード）

アドレッシングモード	対応ビット 5	4	3	2	1	0
Dn	0	0	0	レジスタ番号		
An	0	0	1	レジスタ番号		
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1
d16(PC)	1	1	1	0	1	0
d8(PC,IX)	1	1	1	0	1	1
# Imm	1	1	1	1	0	0

◉サンプル・リスト

```
ADDA.L  AØ,A6
ADDA.L  (A2),A5
```

22 ◉ADDI [Add Immediate　イミディエイト・データの加算]

ADDI {.B／.W／.L} ＃〈data〉,〈ea〉	X	N	Z	V	C
	*	*	*	*	*

解説

ディスティネーション・オペランドとイミディエイト値を加算し，結果をディスティネーションへ格納します．

イミディエイト値のサイズは，オペレーションサイズと同じでなければならず，指定したサイズに入りきらない大きさのイミディエイト値は指定できません．

CCR

X：Cビットと同じ値
N：演算結果が負ならセット(1)，それ以外はリセット(0)
Z：演算結果がゼロならセット(1)，それ以外はリセット(0)
V：オーバフローが発生すればセット(1)，それ以外はリセット(0)
C：桁上がり（キャリ）が発生すればセット(1)，それ以外はリセット(0)

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	サイズ		実効アドレス モード			実効アドレス レジスタ		

サイズ	対応ビット 7	6
バイト	0	0
ワード	0	1
ロングワード	1	0

実効アドレス（データ・可変モード）

アドレッシングモード	対応ビット 5	4	3	2	1	0
Dn	0	0	0	レジスタ番号		
(An)	0	1	0	レジスタ番号		
(An)＋	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1

〈第2ワード：サイズがバイトおよびワードの時〉

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

ビット7〜0：サイズがバイトの時に使用され、イミディエイト値の8ビットが格納される．

ビット15〜0：サイズがワードの時に使用され、指定したワードのイミディエイト値が格納される．

〈第2および第3ワード：ロングワードサイズを指定した時〉

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
指定したイミディエイト値の上位ワード															
指定したイミディエイト値の下位ワード															

◉アドレッシング・モード

sou	size	dest															
		Dn		(An)		(An)+		−(An)		d16(An)		d8(An,IX)		Abs.W		Abs.L	
		#	~	#	~	#	~	#	~	#	~	#	~	#	~	#	~
#Imm	B	4	8	4	16	4	16	4	18	6	20	6	22	6	20	8	24
	W	4	8	4	16	4	16	4	18	6	20	6	22	6	20	8	24
	L	6	16	6	28	6	28	6	30	8	32	8	34	8	32	10	36

◉サンプル・リスト

```
ADDI.L  #$EAØ41,D7
```

23◉ADDQ [Add Quick クイック加算]

ADDQ {.B／.W／.L} ＃〈data〉,〈ea〉	X	N	Z	V	C
	*	*	*	*	*

解説

ディスティネーション・オペランドとイミディエイト値を加算し,結果をディスティネーションのロケーションである〈ea〉へ格納します.

イミディエイト値として指定できる範囲は1～8までに限定されます.

〈ea〉にAn(アドレスレジスタ直接形式)を指定した場合,以下の点に注意が必要です.

- オペレーションサイズはワードまたはロングワードであり，バイトサイズは許されない.
- オペレーションサイズに関係なく，ディスティネーションに指定したアドレスレジスタの全32ビットが使用される.
- CCRは本命令の影響を受けない.

CCR

X：Cビットと同じ値
N：演算結果が負ならセット(1)，それ以外はリセット(0)
Z：演算結果がゼロならセット(1)，それ以外はリセット(0)
V：オーバフローが発生すればセット(1)，それ以外はリセット(0)
C：桁上がり（キャリ）が発生すればセット(1)，それ以外はリセット(0)

◉アドレッシング・モード

sou	size	dest																	
		Dn		An		(An)		(An)+		−(An)		d16(An)		d8(An,IX)		Abs.W		Abs.L	
		＃	～	＃	～	＃	～	＃	～	＃	～	＃	～	＃	～	＃	～	＃	～
＃Imm	B	2	4			2	12	2	12	2	14	4	16	4	18	4	16	6	20
	W	2	4	2	4	2	12	2	12	2	14	4	16	4	18	4	16	6	20
	L	2	8	2	8	2	20	2	20	2	22	4	24	4	26	4	24	6	28

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	データ			0	サイズ		実効アドレス					
										モード			レジスタ		

指定したイミディエイト・データ
000は8を意味し、1～7は001～111に対応する

サイズ	対応ビット	
	7	6
バイト	0	0
ワード	0	1
ロングワード	1	0

実効アドレス（可変モード）

アドレッシングモード	対応ビット					
	5	4	3	2	1	0
Dn	0	0	0	レジスタ番号		
An*	0	0	1	レジスタ番号		
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1

*バイトサイズ不可

◉サンプル・リスト

```
ADDQ.L  #8,A2
ADDQ.W  #4,DØ
```

24 ◉ADDX [Add Extended 拡張加算]

ADDX {.B／.W／.L} Dn, Dn	X	N	Z	V	C
	*	*	*	*	*

解説　ディスティネーション・オペランドにソース・オペランドとXビットを加算し，結果をディスティネーションで指定したDnへ格納します．

CCRのZビットは，通常，演算を実行する前にプログラムでセットされるため，このビットを使用して，多倍精度演算を終了したときの演算結果がゼロであるかをテストできます．

CCR

X：Cビットと同じ値
N：演算結果が負ならセット(1)，それ以外はリセット(0)
Z：演算結果がゼロでなければリセット(0)，それ以外は変化せず
V：オーバフローが発生すればセット(1)，それ以外はリセット(0)
C：桁上がり（キャリ）が発生すればセット(1)，それ以外はリセット(0)

◉アドレッシング・モード

sou	size	dest Dn #	dest Dn ～
Dn	B	2	4
	W	2	4
	L	2	8

◉機械語フォーマット

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	1	0	1	ディスティネーションレジスタ	1	サイズ	0	0	R/M	ソースレジスタ

ソースレジスタ → ソース・オペランドで指定したデータレジスタ番号（000～111）

R/M → 0：データレジスタとデータレジスタとの操作

サイズ →

サイズ	対応ビット 7	対応ビット 6
バイト	0	0
ワード	0	1
ロングワード	1	0

ディスティネーションレジスタ → ディスティネーション・オペランドで指定したデータレジスタ番号（000～111）

◉サンプル・リスト

```
ADDX.L  D4,D7
```

25◉ADDX [Add Extended 拡張加算]

ADDX {.B／.W／.L} −(An), −(An)	X	N	Z	V	C
	*	*	*	*	*

解説　ディスティネーション・オペランドにソース・オペランドとXビットを加算し，結果をディスティネーションへ格納します．

CCRのZビットは，通常，演算を実行する前にプログラムでセットされるため，このビットを使用して，多倍精度演算を終了したときの演算結果がゼロであるかをテストできます．

CCR

X：Cビットと同じ値

N：演算結果が負ならセット(1)，それ以外はリセット(0)

Z：演算結果がゼロでなければリセット(0)，それ以外は変化せず

V：オーバフローが発生すればセット(1)，それ以外はリセット(0)

C：桁上がり（キャリ）が発生すればセット(1)，それ以外はリセット(0)

◉アドレッシング・モード

sou	size	dest −(An) #	dest −(An) ~
−(An)	B	2	18
	W	2	18
	L	2	30

◉機械語フォーマット

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	1	0	1	ディスティネーションレジスタ	1	サイズ	0	0	R/M	ソースレジスタ

ソースレジスタ → ソース・オペランドで指定したアドレスレジスタ番号（000～111）

R/M → 1：メモリとメモリとの操作

サイズ →

サイズ	対応ビット 7	対応ビット 6
バイト	0	0
ワード	0	1
ロングワード	1	0

ディスティネーションレジスタ → ディスティネーション・オペランドで指定したアドレスレジスタ番号（000～111）

◉サンプル・リスト

```
ADDX.W  -(A1),-(A2)
```

26◉SUB [Subtract binary 2進減算]

SUB {.B/.W/.L} Dn, 〈ea〉	X	N	Z	V	C
	*	*	*	*	*

解説 ディスティネーション・オペランドからソース・オペランドを減算し，結果をディスティネーションへ格納します．

CCR
X：Cビットと同じ値
N：演算結果が負ならセット(1)，それ以外はリセット(0)
Z：演算結果がゼロならセット(1)，それ以外はリセット(0)
V：オーバフローが発生すればセット(1)，それ以外はリセット(0)
C：桁下がり（ボロー）が発生すればセット(1)，それ以外はリセット(0)

◉アドレッシング・モード

sou	size	dest (An) #	(An) ~	(An)+ #	(An)+ ~	−(An) #	−(An) ~	d16(An) #	d16(An) ~	d8(AnIX) #	d8(AnIX) ~	Abs.W #	Abs.W ~	Abs.L #	Abs.L ~
Dn	B	2	12	2	12	2	14	4	16	4	18	4	16	6	20
	W	2	12	2	12	2	14	4	16	4	18	4	16	6	20
	L	2	20	2	20	2	22	4	24	4	26	4	24	6	28

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	レジスタ			OPモード			実効アドレス モード			実効アドレス レジスタ		

サイズ	対応ビット 8	7	6
バイト	1	0	0
ワード	1	0	1
ロングワード	1	1	0

実効アドレス（メモリ・可変モード）

アドレッシングモード	対応ビット 5	4	3	2	1	0
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1

→ 指定したデータレジスタ番号（000～111）

◉サンプル・リスト

```
SUB.W   D2,(A1)
SUB.L   D3,(AØ)+
```

27 ◉SUB [Subtract binary　2進減算]

SUB {.B/.W/.L} <ea>, Dn	X	N	Z	V	C
	*	*	*	*	*

解説

ディスティネーション・オペランドからソース・オペランドを減算し，結果をディスティネーションへ格納します．

ソース・オペランドの<ea>にAn（アドレスレジスタ直接）を指定する場合，ワードまたはロングワードのオペレーションサイズがサポートされ，バイトサイズを指定できません．

CCR

X：Cビットと同じ値
N：演算結果が負ならセット(1)，それ以外はリセット(0)
Z：演算結果がゼロならセット(1)，それ以外はリセット(0)
V：オーバフローが発生すればセット(1)，それ以外はリセット(0)
C：桁下がり（ボロー）が発生すればセット(1)，それ以外はリセット(0)

◉アドレッシング・モード

sou	size	dest Dn #	dest Dn ~
Dn	B	2	4
	W	2	4
	L	2	8
An	B		
	W	2	4
	L	2	8
(An)	B	2	8
	W	2	8
	L	2	14
(An)+	B	2	8
	W	2	8
	L	2	14
−(An)	B	2	10
	W	2	10
	L	2	16
d16(An)	B	4	12
	W	4	12
	L	4	18
d8(An,IX)	B	4	14
	W	4	14
	L	4	20
Abs.W	B	4	12
	W	4	12
	L	4	18
Abs.L	B	6	16
	W	6	16
	L	6	22
d16(PC)	B	4	12
	W	4	12
	L	4	18
d8(PC,IX)	B	4	14
	W	4	14
	L	4	20
#Imm (注)	B	4	8
	W	4	8
	L	6	14

◉機械語フォーマット

15	14	13	12	11 10 9	8 7 6	5 4 3	2 1 0
1	0	0	1	レジスタ	OPモード	実効アドレス モード	実効アドレス レジスタ

指定したデータレジスタ番号（000～111）

サイズ	対応ビット 8	7	6
バイト	0	0	0
ワード	0	0	1
ロングワード	0	1	0

実効アドレス（すべてのモード）

アドレッシングモード	5	4	3	2	1	0
Dn	0	0	0	レジスタ番号		
An*	0	0	1	レジスタ番号		
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	1	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1
d16(PC)	1	1	1	0	1	0
d8(PC,IX)	1	1	1	0	1	1
(注) # Imm	1	1	1	1	0	0

＊バイトサイズ不可
（注）SUBIを使用する

◉サンプル・リスト

```
SUB.L   (AØ)+,DØ
SUB.B   12(A1,A2.L),D2
```

28◉SUBA [Subtract Address アドレス・データの減算]

SUBA {.W／.L} <ea>, An

X	N	Z	V	C
—	—	—	—	—

解説

指定したアドレスレジスタからソース・オペランドを減算し，結果をアドレスレジスタへ格納します．

オペレーションサイズは，ワードまたはロングワードがサポートされ，バイトサイズは許されません。さらに，ワードのオペレーションでは，ソース・オペランドは32ビットに符号拡張され，アドレスレジスタの全32ビットが演算の対象となります．

CCR

X：変化せず
N：変化せず
Z：変化せず
V：変化せず
C：変化せず

◉アドレッシング・モード

sou	size	dest An #	dest An ～
Dn	B		
	W	2	8
	L	2	8
An	B		
	W	2	8
	L	2	8
(An)	B		
	W	2	12
	L	2	14
(An)+	B		
	W	2	12
	L	2	14
−(An)	B		
	W	2	14
	L	2	16
d16(An)	B		
	W	4	16
	L	4	18
d8(An,IX)	B		
	W	4	18
	L	4	20
Abs.W	B		
	W	4	16
	L	4	18
Abs.L	B		
	W	6	20
	L	6	22
d16(PC)	B		
	W	4	16
	L	4	18
d8(PC,IX)	B		
	W	4	18
	L	4	20
#Imm	B		
	W	4	12
	L	6	14

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	レジスタ			OPモード			実効アドレス モード			実効アドレス レジスタ		

ディスティネーション・オペランドで指定したアドレスレジスタ番号（000～111）

サイズ	対応ビット 8	7	6
ワード	0	1	1
ロングワード	1	1	1

実効アドレス（すべてのモード）

アドレッシングモード	対応ビット 5	4	3	2	1	0
Dn	0	0	0	レジスタ番号		
An	0	0	1	レジスタ番号		
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1
d16(PC)	1	1	1	0	1	0
d8(PC,IX)	1	1	1	0	1	1
# Imm	1	1	1	1	0	0

◉サンプル・リスト

```
SUBA.L  A2,A1
SUBA.L  $2000,A4
SUBA.L  #$FAD00,A5
```

29 ◉SUBI [Subtract Immediate イミディエイト・データとの減算]

SUBI {.B/.W/.L} ＃ <data>, <ea>	X	N	Z	V	C
	*	*	*	*	*

解説　ディスティネーション・オペランドからイミディエイト値を減算し，結果をディスティネーションへ格納します．

イミディエイト値のサイズは，オペレーションサイズと同じでなければならず，指定したサイズに入りきらない大きさのイミディエイト値は指定できません．

CCR

X：Cビットと同じ値

N：演算結果が負ならセット(1)，それ以外はリセット(0)

Z：演算結果がゼロならセット(1)，それ以外はリセット(0)

V：オーバフローが発生すればセット(1)，それ以外はリセット(0)

C：桁下がり（ボロー）が発生すればセット(1)，それ以外はリセット(0)

◉アドレッシング・モード

sou	size	dest Dn #	Dn ～	(An) #	(An) ～	(An)+ #	(An)+ ～	−(An) #	−(An) ～	d16(An) #	d16(An) ～	d8(An,IX) #	d8(An,IX) ～	Abs.W #	Abs.W ～	Abs.L #	Abs.L ～
＃Imm	B	4	8	4	16	4	16	4	18	6	20	6	22	6	20	8	24
	W	4	8	4	16	4	16	4	18	6	20	6	22	6	20	8	24
	L	6	16	6	28	6	28	6	30	8	32	8	34	8	32	10	36

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	サイズ		実効アドレス モード			レジスタ		

サイズ	対応ビット 7	6
バイト	0	0
ワード	0	1
ロングワード	1	0

実効アドレス（データ・可変モード）

アドレッシングモード	対応ビット 5	4	3	2 1 0
Dn	0	0	0	レジスタ番号
(An)	0	1	0	レジスタ番号
(An)+	0	1	1	レジスタ番号
−(An)	1	0	0	レジスタ番号
d16(An)	1	0	1	レジスタ番号
d8(An,IX)	1	1	0	レジスタ番号
Abs.W	1	1	1	0 0 0
Abs.L	1	1	1	0 0 1

<第2ワード，バイトまたはワードサイズ時>

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

ビット7〜0 → イミディエイト値の下位バイト（サイズがバイトの時）

ビット15〜0 → イミディエイト値のワード（サイズがワードの時）

<第2，第3ワード，ロングワード時>

15 〜 0
指定したイミディエイト値の上位ワード
指定したイミディエイト値の下位ワード

◉サンプル・リスト

```
SUBI.W  #10,D0
SUBI.L  #$FF00AF,(A2)
```

30◉SUBQ [Subtract　Quick　クイック減算]

SUBQ {.B／.W／.L} ＃ <data>, <ea>	X	N	Z	V	C
	*	*	*	*	*

解説

ディスティネーション・オペランドからイミディエイト値を減算し，結果をディスティネーションへ格納します．

指定できるイミディエイト値は，１～８までの範囲に限定されます．

ディスティネーションの〈ea〉にAn（アドレスレジスタ直接形式）を指定した場合，次の点に注意する必要があります．

- オペレーションサイズは，ワードまたはロングワードがサポートされ，バイトサイズを指定できない．
- オペレーションサイズがワードサイズである場合，演算実行前に32ビットに符号拡張され，アドレスレジスタの全32ビットが使用される．
- CCRは影響を受けない．

CCR

X：Cビットと同じ値

N：演算結果が負ならセット(1)，それ以外はリセット(0)

Z：演算結果がゼロならセット(1)，それ以外はリセット(0)

V：オーバフローが発生すればセット(1)，それ以外はリセット(0)

C：桁下がり（ボロー）が発生すればセット(1)，それ以外はリセット(0)

◉アドレッシング・モード

sou	size	dest Dn ＃	Dn ～	An ＃	An ～	(An) ＃	(An) ～	(An)+ ＃	(An)+ ～	−(An) ＃	−(An) ～	d16(An) ＃	d16(An) ～	d8(An,IX) ＃	d8(An,IX) ～	Abs.W ＃	Abs.W ～	Abs.L ＃	Abs.L ～
＃Imm	B	2	4			2	12	2	12	2	14	4	16	4	18	4	16	6	20
	W	2	4	2	4	2	12	2	12	2	14	4	16	4	18	4	16	6	20
	L	2	8	2	8	2	16	2	16	2	22	4	24	4	26	4	24	6	28

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	データ			1	サイズ		実効アドレス モード			実効アドレス レジスタ		

サイズ	対応ビット 7	6
バイト	0	0
ワード	0	1
ロングワード	1	0

実効アドレス（可変モード）

アドレッシングモード	対応ビット 5	4	3	2　1　0
Dn	0	0	0	レジスタ番号
An*	0	0	1	レジスタ番号
(An)	0	1	0	レジスタ番号
(An)+	0	1	1	レジスタ番号
−(An)	1	0	0	レジスタ番号
d16(An)	1	0	1	レジスタ番号
d8(An,IX)	1	1	0	レジスタ番号
Abs.W	1	1	1	0　0　0
Abs.L	1	1	1	0　0　1

＊バイトサイズ不可

イミディエイトデータ０００は８を意味し，００１～１１１は１～７に対応する．

◉サンプル・リスト

```
SUBQ.W   #8,4(AØ,D2.L)
SUBQ.L   #5,DØ
```

31 ◉SUBX [Subtract with Extend 拡張付き減算]

SUBX {.B／.W／.L} Dn, Dn	X	N	Z	V	C
	*	*	*	*	*

解説　ディスティネーション・オペランドからソース・オペランドとXビットを減算し，結果をディスティネーションへ格納します．

CCRのZビットは，通常，演算を実行する前にプログラムでセットされるため，このビットを使用して，多倍精度演算を終了したときの演算結果がゼロであるかをテストできます．

CCR

X：Cビットと同じ値

N：演算結果が負ならセット(1)，それ以外はリセット(0)

Z：演算結果がゼロでなければリセット(0)，それ以外は変化せず

V：オーバフローが発生すればセット(1)，それ以外はリセット(0)

C：桁下がり（ボロー）が発生すればセット(1)，それ以外はリセット(0)

◉アドレッシング・モード

sou	size	dest Dn #	dest Dn ～
Dn	B	2	4
	W	2	4
	L	2	8

◉機械語フォーマット

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	0	0	1	ディスティネーションレジスタ	1	サイズ	0	0	R/M	ソースレジスタ

ソースレジスタ → ソース・オペランドで指定したデータレジスタ番号（000～111）

R/M → 0：データレジスタとデータレジスタの操作

サイズ →

サイズ	対応ビット 7	6
バイト	0	0
ワード	0	1
ロングワード	1	0

ディスティネーションレジスタ → ディスティネーション・オペランドで指定したデータレジスタ番号（000～111）

◉サンプル・リスト

```
SUBX.B  D1,D2
```

32◉SUBX [Subtract with Extend　拡張付き減算]

SUBX {.B／.W／.L}　－(An), －(An)	X	N	Z	V	C
	*	*	*	*	*

解説

ディスティネーション・オペランドからソース・オペランドとXビットを減算し，結果をディスティネーションへ格納します．

CCRのZビットは，通常，演算を実行する前にプログラムでセットされるため，このビットを使用して，多倍精度演算を終了したときの演算結果がゼロであるかをテストできます．

CCR

X：Cビットと同じ値

N：演算結果が負ならセット(1)，それ以外はリセット(0)

Z：演算結果がゼロでなければリセット(0)，それ以外は変化せず

V：オーバフローが発生すればセット(1)，それ以外はリセット(0)

C：桁下がり（ボロー）が発生すればセット(1)，それ以外はリセット(0)

◉アドレッシング・モード

sou	size	dest －(An) #	dest －(An) ～
－(An)	B	2	8
	W	2	8
	L	2	30

◉機械語フォーマット

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	0	0	1	ディスティネーションレジスタ	1	サイズ	0	0	R/M	ソースレジスタ

ソース・オペランドで指定したアドレスレジスタ番号（０００～１１１）

R/M　1：メモリとメモリとの操作

サイズ	対応ビット 7	対応ビット 6
バイト	0	0
ワード	0	1
ロングワード	1	0

ディスティネーション・オペランドで指定したアドレスレジスタ番号（０００～１１１）

◉サンプル・リスト

```
SUBX.L  -(A1),-(A5)
```

33◉MULS [Multiply as Signed　符号付き乗算]

MULS {.W}　<ea>, Dn

X	N	Z	V	C
—	*	*	0	0

解説

2つの符号付き16ビット整数（−32768〜32767）のオペランドを乗算し，結果は，32ビットの符号付き整数としてディスティネーションへ格納します．

ディスティネーション・オペランドとしてのDnは，演算時に下位16ビットが取り出され上位ワードは無視されますが，結果は32ビットとして格納されることから，元の内容の32ビットすべてが失われます．

CCR

X：変化せず
N：演算結果が負ならセット(1)，それ以外はリセット(0)
Z：演算結果がゼロならセット(1)，それ以外はリセット(0)
V：常にリセット(0)
C：常にリセット(0)

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	レジスタ			1	1	1	実効アドレス					
										モード			レジスタ		

レジスタ：指定したデータレジスタ番号（000〜111）

実効アドレス（データモード）

アドレッシングモード	対応ビット 5	4	3	2	1	0
Dn	0	0	0	レジスタ番号		
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1
d16(PC)	1	1	1	0	1	0
d8(PC,IX)	1	1	1	0	1	1
# Imm	1	1	1	1	0	0

◉アドレッシング・モード

sou	size	dest Dn #	~
Dn	B		
	W	2	<70
	L		
(An)	B		
	W	2	<74
	L		
(An)+	B		
	W	2	<74
	L		
−(An)	B		
	W	2	<76
	L		
d16(An)	B		
	W	4	<78
	L		
d8(An,IX)	B		
	W	4	<80
	L		
Abs.W	B		
	W	4	<78
	L		
Abs.L	B		
	W	6	<82
	L		
d16(PC)	B		
	W	4	<78
	L		
d8(PC,IX)	B		
	W	4	<80
	L		
# Imm	B		
	W	4	<74
	L		

◉サンプル・リスト

```
MULS    (A2),DØ
MULS    4(A4,A5.L),D3
```

34●MULU [Multiply as Unsigned　符号なし乗算]

MULU {.W}　〈ea〉, Dn	X	N	Z	V	C
	—	＊	＊	0	0

解説

2つの符号なし16ビット整数（0～65535）のオペランドを乗算し，結果は，32ビットの符号なし整数としてディスティネーションへ格納します．

ディスティネーション・オペランドとしてのDnは，演算時には下位16ビットが取り出され上位ワードは無視されますが，結果は32ビットとして格納されることから，元の内容の32ビットすべてが失われます．

CCR

X：変化せず
N：演算の結果，データの最上位ビット（MSB）が"1"ならセット(1)，それ以外はリセット(0)
Z：演算結果がゼロならセット(1)，それ以外はリセット(0)
V：常にリセット(0)
C：常にリセット(0)

●アドレッシング・モード

sou	size	dest Dn #	dest Dn ～
Dn	B		
	W	2	<70
	L		
(An)	B		
	W	2	<74
	L		
(An)＋	B		
	W	2	<74
	L		
－(An)	B		
	W	2	<76
	L		
d16(An)	B		
	W	4	<78
	L		
d8(An,IX)	B		
	W	4	<80
	L		
Abs.W	B		
	W	4	<78
	L		
Abs.L	B		
	W	6	<82
	L		
d16(PC)	B		
	W	4	<78
	L		
d8(PC,IX)	B		
	W	4	<80
	L		
＃Imm	B		
	W	4	<74
	L		

●機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	レジスタ			0	1	1	実効アドレス モード			実効アドレス レジスタ		

レジスタ → 指定したデータレジスタ番号（000～111）

実効アドレス → 実効アドレス（データモード）

アドレッシングモード	対応ビット 5	4	3	2	1	0
Dn	0	0	0	レジスタ番号		
(An)	0	1	0	レジスタ番号		
(An)＋	0	1	1	レジスタ番号		
－(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1
d16(PC)	1	1	1	0	1	0
b8(PC,IX)	1	1	1	0	1	1
＃Imm	1	1	1	1	0	0

●サンプル・リスト

```
MULU    D2,D3
MULU    (A3),D0
```

35◉DIVS [Divide as Signed 符号付き除算]

DIVS {.W} 〈ea〉, Dn

X	N	Z	V	C
—	*	*	*	0

解説　ディスティネーション・オペランドをソース・オペランドで除算し，結果をディスティネーションへ格納します．

演算は符号付き算術演算で行われ，オペランドに関する要点は次の通りです．

- ●ソース・オペランドである〈ea〉の内容はワードであり，下位16ビットが演算に使用される．
- ●ディスティネーション・オペランドのDnはロングワードであり，32ビットのすべてが演算に使用される．

演算結果に関する要点は次の通りです．

- ●商はディスティネーション・オペランドDnの下位ワードへ格納される．
- ●余りはディスティネーション・オペランドDnの上位ワードへ格納される。
- ●余りがゼロでない限り，その符号は被除数と同じ．

本命令では，次の２つの特別な実行結果があります．

- ●ゼロで割るとTRAP（トラップ）が発生し，プロセッサは例外処理を開始する．
- ●命令完了前にオーバーフローを検出すると，Vビットがセットされ，除算は実行されず，ディスティネーション・オペランドDnは更新されない．つまり，商も余りも格納されない．

◉アドレッシング・モード

sou	size	dest Dn #	dest Dn ~
Dn	B		
	W	2	<158
	L		
(An)	B		
	W	2	<162
	L		
(An)+	B		
	W	2	<162
	L		
−(An)	B		
	W	2	<164
	L		
d16(An)	B		
	W	4	<166
	L		
d8(An,IX)	B		
	W	4	<168
	L		
Abs.W	B		
	W	4	<166
	L		
Abs.L	B		
	W	6	<170
	L		
d16(PC)	B		
	W	4	<166
	L		
d8(PC.IX)	B		
	W	4	<168
	L		
#Imm	B		
	W	4	<162
	L		

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	レジスタ			1	1	1	実効アドレス モード			実効アドレス レジスタ		

レジスタ：指定したデータレジスタ番号（０００～１１１）

実効アドレス（データモード）

アドレッシングモード	対応ビット 5	4	3	2	1	0
Dn	0	0	0	レジスタ番号		
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Ads.L	1	1	1	0	0	1
d16(PC)	1	1	1	0	1	0
d8(PC,IX)	1	1	1	0	1	1
# Imm	1	1	1	1	0	0

CCR

X：変化せず
N：商が負ならセット(1)，それ以外はリセット(0)
Z：商がゼロならセット(1)，それ以外はリセット(0)
V：オーバフローが発生すればセット(1)，それ以外はリセット(0)
C：常にリセット(0)
(注)オーバフローが発生した場合，NビットおよびZビットは意味を持たず，除算結果を正しく反映しない.

◉サンプル・リスト

```
DIVS    DØ,D1
DIVS    (AØ)+,D3
```

36◉DIVU [Divide as Unsigned　符号なし除算]

DIVU {.W} 〈ea〉, Dn

X	N	Z	V	C
—	*	*	*	0

解説　ディスティネーション・オペランドをソース・オペランドで除算し，結果をディスティネーションへ格納します．

演算は符号なし算術演算で行われ，オペランドに関する要点は次の通りです．
- ソース・オペランドである〈ea〉の内容はワードであり，下位16ビットが演算に使用される．
- ディスティネーション・オペランドのDnはロングワードであり，32ビットのすべてが演算に使用される．

演算結果に関する要点は次の通りです．
- 商はディスティネーション・オペランドDnの下位ワードへ格納される．
- 余りはディスティネーション・オペランドDnの上位ワードへ格納される．
- 余りがゼロでない限り，その符号は被除数と同じである．

本命令では，次の2つの特別な実行結果があります．
- ゼロで割るとTRAP（トラップ）が発生し，プロセッサは例外処理を開始する．
- 命令完了前にオーバフローを検出すると，Vビットがセットされ，除算は実行されず，ディスティネーション・オペランド Dnは更新されない．つまり，商も余りも格納されない．

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	レジスタ			0	1	1	実効アドレス					
										モード			レジスタ		

レジスタ → 指定したデータレジスタ番号（０００～１１１）

実効アドレス（データモード）

アドレッシングモード	対応ビット 5	4	3	2	1	0
Dn	0	0	0	レジスタ番号		
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An, IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1
d16(PC)	1	1	1	0	1	0
d8(PC, IX)	1	1	1	0	1	1
＃ Imm	1	1	1	1	0	0

CCR

X：変化せず
N：商の最上位ビットが〝1〞ならセット(1)，それ以外はリセット(0)
Z：商がゼロならセット(1)，それ以外はリセット(0)
V：オーバフローが発生すればセット(1)，それ以外はリセット(0)
C：常にリセット(0)

(注)オーバフローが発生した場合，NビットおよびZビットは意味を持たず，除算結果を正しく反映しない．

◉アドレッシング・モード

sou	size	dest Dn #	dest Dn ～
Dn	B		
	W	2	<140
	L		
(An)	B		
	W	2	<144
	L		
(An)＋	B		
	W	2	<144
	L		
−(An)	B		
	W	2	<146
	L		
d16(An)	B		
	W	4	<148
	L		
d8(An,IX)	B		
	W	4	<150
	L		
Abs.W	B		
	W	4	<148
	L		
Abs.L	B		
	W	6	<152
	L		
d16(PC)	B		
	W	4	<148
	L		
d8(PC,IX)	B		
	W	4	<150
	L		
＃Imm	B		
	W	4	<144
	L		

◉サンプル・リスト

```
DIVU    D2,D3
DIVU    (AØ),D4
DIVU    $2ØØØ,D4
```

37 ◉CMP [Compare 比較]

CMP {.B/.W/.L} 〈ea〉, Dn

X	N	Z	V	C
—	*	*	*	*

解説 ディスティネーション・オペランドからソース・オペランドを減算し，その結果がCCRへ反映されますが，ディスティネーション・オペランドDnの内容は変更されません（減算結果は格納されない）．

CCR
X：変化せず
N：演算結果が負ならセット(1)，それ以外はリセット(0)
Z：演算結果がゼロならセット(1)，それ以外はリセット(0)
V：オーバフローが発生すればセット(1)，それ以外はリセット(0)
C：桁下がり（ボロー）が発生すればセット(1)，それ以外はリセット(0)

◉機械語フォーマット

15	14	13	12	11 10 9	8 7 6	5 4 3	2 1 0
1	0	1	1	レジスタ	OPモード	実効アドレス モード	実効アドレス レジスタ

レジスタ：指定したレジスタ番号（000～111）

サイズ	対応ビット 8	7	6
バイト	0	0	0
ワード	0	0	1
ロングワード	0	1	0

実効アドレス（すべてのモード）

アドレッシングモード	対応ビット 5	4	3	2	1	0
Dn	0	0	0	レジスタ番号		
An*	0	0	1	レジスタ番号		
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1
d16(PC)	1	1	1	0	1	0
d8(PC,IX)	1	1	1	0	1	1
(注)# Imm	1	1	1	1	0	0

* バイトサイズ不可
(注) CMPIを使用する

◉サンプル・リスト

```
CMP     D2,D3
CMP     $2000,D4
CMP.L   $3A(A2,D4.L),D2
```

◉アドレッシング・モード

sou	size	dest Dn #	dest Dn ～
Dn	B	2	4
	W	2	4
	L	2	6
An	B		
	W	2	4
	L	2	6
(An)	B	2	8
	W	2	8
	L	2	14
(An)+	B	2	8
	W	2	8
	L	2	14
−(An)	B	2	10
	W	2	10
	L	2	16
d16(An)	B	4	12
	W	4	12
	L	4	18
d8(An,IX)	B	4	14
	W	4	14
	L	4	20
Abs.W	B	4	12
	W	4	12
	L	4	18
Abs.L	B	6	16
	W	6	16
	L	6	22
d16(PC)	B	4	12
	W	4	12
	L	4	18
d8(PC,IX)	B	4	14
	W	4	14
	L	4	20
#Imm (注)	B	4	8
	W	4	8
	L	6	14

38◉CMPA [Compare Address アドレス・データ比較]

CMPA {.W／.L} <ea>, An	X	N	Z	V	C
	—	*	*	*	*

解説

ディスティネーション・オペランドからソース・オペランドを減算し，その結果がCCRへ反映されますが，ディスティネーション・オペランドの内容は変更されません（減算結果は格納されない）．

オペランドサイズがワードである場合，ソースオペランドである〈ea〉の内容は，演算実行前に32ビットに符号拡張され，アドレスレジスタの全32ビットが演算の対象となります．

CCR

X：変化せず
N：演算結果が負ならセット(1)，それ以外はリセット(0)
Z：演算結果がゼロならセット(1)，それ以外はリセット(0)
V：オーバフローが発生すればセット(1)，それ以外はリセット(0)
C：桁下がり（ボロー）が発生すればセット(1)，それ以外はリセット(0)

◉アドレッシング・モード

sou	size	dest An #	dest An ~
Dn	B		
	W	2	6
	L	2	6
An	B		
	W	2	6
	L	2	6
(An)	B		
	W	2	10
	L	2	14
(An)+	B		
	W	2	10
	L	2	14
−(An)	B		
	W	2	12
	L	2	16
d16(An)	B		
	W	4	14
	L	4	18
d8(An,IX)	B		
	W	4	16
	L	4	20
Abs.W	B		
	W	4	12
	L	4	18
Abs.L	B		
	W	6	18
	L	6	22
d16(PC)	B		
	W	4	14
	L	4	18
d8(PC,IX)	B		
	W	4	16
	L	4	20
#Imm	B		
	W	4	10
	L	6	14

◉機械語フォーマット

15	14	13	12	11 10 9	8 7 6	5 4 3	2 1 0
						実効アドレス	
1	0	1	1	レジスタ	OPモード	モード	レジスタ

レジスタ → 指定したアドレスレジスタ番号(000～111)

サイズ	対応ビット 8	7	6
ワード	0	1	1
ロングワード	1	1	1

実効アドレス（すべてのモード）

アドレッシングモード	5	4	3	2	1	0
Dn	0	0	0	レジスタ番号		
An	0	0	1	レジスタ番号		
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1
d16(PC)	1	1	1	0	1	0
d8(PC,IX)	1	1	1	0	1	1
# Imm	1	1	1	1	0	0

◉サンプル・リスト

```
CMPA.L  A1,A2
CMPA.L  $2000,A4
CMPA.L  (A2),A6
```

39 ◉CMPI [Compare Immediate イミディエイト・データとの比較]

CMPI {.B/.W/.L} # <data>, <ea>	X	N	Z	V	C
	—	*	*	*	*

解説 ディスティネーション・オペランドからイミディエイト値を減算し，その結果がCCRへ反映されますが，ディスティネーション・オペランドは変更されません（減算結果は格納されない）.

イミディエイト値のサイズは，オペレーションサイズと同一です.

CCR

X：変化せず

N：演算結果が負ならセット(1)，それ以外はリセット(0)

Z：演算結果がゼロならセット(1)，それ以外はリセット(0)

V：オーバフローが発生すればセット(1)，それ以外はリセット(0)

C：桁下がり（ボロー）が発生すればセット(1)，それ以外はリセット(0)

◉アドレッシング・モード

sou	size	dest															
		Dn		(An)		(An)+		−(An)		d16(An)		d8(An,IX)		Abs.W		Abs.L	
		#	~	#	~	#	~	#	~	#	~	#	~	#	~	#	~
#Imm	B	4	8	4	12	4	12	4	14	6	16	6	18	6	16	8	20
	W	4	8	4	12	4	12	4	14	6	16	6	18	6	16	8	20
	L	6	14	6	20	6	20	6	22	8	24	8	26	8	24	10	28

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	0	サイズ		実効アドレス					
										モード			レジスタ		

サイズ	対応ビット	
	7	6
バイト	0	0
ワード	0	1
ロングワード	1	0

実効アドレス（データ・可変モード）

アドレッシングモード	対応ビット					
	5	4	3	2	1	0
Dn	0	0	0	レジスタ番号		
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1

<第2ワード：バイトまたはワードサイズの時>

15 … 8	7 … 0

ビット7〜0：サイズがバイトの時に使用され，指定したイミディエイト値の8ビットが格納される.

ビット15〜0：サイズがワードの時に使用され，指定したイミディエイト値の16ビットが格納される.

<第2および第3ワード：ロングワードサイズの時>

15 … 0
指定したイミディエイト値の上位ワード
指定したイミディエイト値の下位ワード

◉サンプル・リスト

```
CMPI.B  #$4E,(A1)+
CMPI.L  #$FFDA2,D0
CMPI.W  #$FF00,$2000
```

40 ◉CMPM [Compare Memory メモリ比較]

CMPM {.B／.W／.L} (An)＋, (An)＋	X	N	Z	V	C
	—	*	*	*	*

解説　ディスティネーション・オペランドからソース・オペランドを減算し，その結果がCCRへ反映されますが，ディスティネーション・オペランドの内容は変更されません（減算結果は格納されない）.

ソース，ディスティネーションともポストインクリメントによるアドレスレジスタ間接形式でアクセスされ，ある区間（ブロック）を比較あるいは検索するのに便利です．

CCR
X：変化せず
N：演算結果が負ならセット(1)，それ以外はリセット(0)
Z：演算結果がゼロならセット(1)，それ以外はリセット(0)
V：オーバフローが発生すればセット(1)，それ以外はリセット(0)
C：桁下がり（ボロ）が発生すればセット(1)，それ以外はリセット(0)

◉アドレッシング・モード

sou	size	dest (An)＋ #	dest (An)＋ ～
(An)＋	B	2	12
	W	2	12
	L	2	20

◉機械語フォーマット

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	0	1	1	ディスティネーション アドレスレジスタ	1	サイズ	0	0	1	ソース アドレスレジスタ

ソース・オペランドで指定したアドレスレジスタ番号（０００～１１１）

サイズ	対応ビット 7	6
バイト	0	0
ワード	0	1
ロングワード	1	0

ディスティネーション・オペランドで指定したアドレスレジスタ番号（０００～１１１）

◉サンプル・リスト

```
CMPM.W  (A2)+,(A5)+
```

41 ◉CLR [Clear an Operand　オペランドのクリア]

CLR {.B／.W／.L}　<ea>

X	N	Z	V	C
—	0	1	0	0

解説　ディスティネーション・オペランドのすべてのビットをクリア（0）します．
メモリへの操作は，一度読み出された後で書き込まれます．

CCR
X：変化せず
N：常にリセット(0)
Z：常にセット(1)
V：常にリセット(0)
C：常にリセット(0)

◉アドレッシング・モード

sou	size	dest															
		Dn		(An)		(An)+		−(An)		d16(An)		d8(An,IX)		Abs.W		Abs.L	
		#	～	#	～	#	～	#	～	#	～	#	～	#	～	#	～
	B	2	4	2	12	2	12	2	14	4	16	4	18	4	16	6	20
	W	2	4	2	12	2	12	2	14	4	16	4	18	4	16	6	20
	L	2	6	2	20	2	20	2	22	4	24	4	26	4	24	6	28

＊ソース・オペランドは存在しない

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	サイズ		実効アドレス					
										モード			レジスタ		

サイズ	対応ビット	
	7	6
バイト	0	0
ワード	0	1
ロングワード	1	0

実効アドレス（データ・可変モード）

アドレッシングモード	対応ビット					
	5	4	3	2	1	0
Dn	0	0	0	レジスタ番号		
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1

◉サンプル・リスト

```
CLR.L   DØ
CLR.W   (A1)+
CLR.L   $FFØØ
```

42 ◉EXT [Sign Extend 符号拡張]

EXT {.W／.L} Dn	X	N	Z	V	C
	—	＊	＊	0	0

解説 データレジスタの符号ビット（バイトならビット7，ワードならビット15）を上位方向へコピーし，バイトをワードに，あるいはワードをロングワードに符号拡張します．

- オペレーションサイズがワード ……Dnのビット7をビット15〜8にコピーしてワードとする．
- オペレーションサイズがロングワード……Dnのビット15をビット31〜16にコピーしてロングワードとする．

CCR
X：変化せず
N：演算結果が負ならセット(1)，それ以外はリセット(0)
Z：演算結果がゼロならセット(1)，それ以外はリセット(0)
V：常にリセット(0)
C：常にリセット(0)

◉アドレッシング・モード

sou	size	dest Dn #	dest Dn ～
	B	2	4
	W	2	4
	L	2	4

＊ソース・オペランドは存在しない

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	OPモード			0	0	0	レジスタ		

レジスタ→指定したデータレジスタ番号（000〜111）

対応ビット 8	7	6	意味
0	1	0	データレジスタの下位バイトをワードに符号拡張
0	1	1	データレジスタの下位ワードをロングワードに符号拡張

◉サンプル・リスト

```
EXT.W   D0
EXT.L   D2
```

43 ◉NEG [Negate　2進符号反転]

NEG {.B／.W／.L}　<ea>

X	N	Z	V	C
*	*	*	*	*

解説　ゼロ（0）からディスティネーション・オペランドを減算し，結果をディスティネーションへ格納します．

CCR
X：Cビットと同じ値
N：演算結果が負ならセット(1)，それ以外はリセット(0)
Z：演算結果がゼロならセット(1)，それ以外はリセット(0)
V：オーバフローが発生すればセット(1)，それ以外はリセット(0)
C：桁下がり（ボロー）が発生すればセット(1)，それ以外はリセット(0)

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	サイズ		実効アドレス					
										モード			レジスタ		

サイズ	対応ビット 7	6
バイト	0	0
ワード	0	1
ロングワード	1	0

実効アドレス（データ・可変モード）

アドレッシングモード	対応ビット 5	4	3	2	1	0
Dn	0	0	0	レジスタ番号		
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1

◉アドレッシング・モード

sou	size	dest Dn #	Dn ~	(An) #	(An) ~	(An)+ #	(An)+ ~	−(An) #	−(An) ~	d16(An) #	d16(An) ~	d8(An,IX) #	d8(An,IX) ~	Abs.W #	Abs.W ~	Abs.L #	Abs.L ~
	B	2	4	2	12	2	12	2	14	4	16	4	18	4	16	6	20
	W	2	4	2	12	2	12	2	14	4	16	4	18	4	16	6	20
	L	2	6	2	20	2	20	2	22	4	24	4	26	4	24	6	28

＊ソース・オペランドは存在しない

◉サンプル・リスト

```
NEG.B    -(AØ)
NEG.L    D3
```

44 ◉NEGX [Negate with Extend 拡張付き2進符号反転]

NEGX {.B/.W/.L} 〈ea〉	X	N	Z	V	C
	*	*	*	*	*

解説

ゼロ（0）からディスティネーション・オペランドとXビットを減算し，結果をディスティネーションのロケーションへ格納します．

CCR

X：Cビットと同じ値
N：演算結果が負ならセット(1)，それ以外はリセット(0)
Z：演算結果がゼロでなければリセット(0)，それ以外は変化せず
V：オーバフローが発生すればセット(1)，それ以外はリセット(0)
C：桁下がり（ボロー）が発生すればセット(1)，それ以外はリセット(0)

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	サイズ		実効アドレス					
										モード			レジスタ		

実効アドレス（データ・可変モード）

アドレッシングモード	実効アドレス	
	モード	レジスタ
Dn	0 0 0	レジスタ番号
(An)	0 1 0	レジスタ番号
(An)+	0 1 1	レジスタ番号
−(An)	1 0 0	レジスタ番号
d16(An)	1 0 1	レジスタ番号
d8(An,IX)	1 1 0	レジスタ番号
Abs.W	1 1 1	0 0 0
Abs.L	1 1 1	0 0 1

サイズ	対応ビット	
	7	6
バイト	0	0
ワード	0	1
ロングワード	1	0

◉アドレッシング・モード

sou	size	dest															
		Dn		(An)		(An)+		−(An)		d16(An)		d8(AnIX)		Abs.W		Abs.L	
		#	~	#	~	#	~	#	~	#	~	#	~	#	~	#	~
	B	2	4	2	12	2	12	2	14	4	16	4	18	4	16	6	20
	W	2	4	2	12	2	12	2	14	4	16	4	18	4	16	6	20
	L	2	6	2	20	2	20	2	22	4	24	4	26	4	24	6	28

＊ソース・オペランドは存在しない

◉サンプル・リスト

```
NEGX.B  DØ
NEGX.W  (AØ)
NEGX.L  (A2)+
```

45◉TST [Test an operand オペランドのテスト]

TST {.B/.W/.L} <ea>	X	N	Z	V	C
	—	*	*	0	0

解説

ディスティネーション・オペランドをゼロ（0）と比較し，結果をCCRへ反映しますが，オペランドは変更されません．

フラグの変化にみられるように，ある値の符号（正負）やゼロか否かを知りたい場合に使います．

CCR

X：変化せず
N：演算結果が負ならセット(1)，それ以外はリセット(0)
Z：演算結果がゼロならセット(1)，それ以外はリセット(0)
V：常にリセット(0)
C：常にリセット(0)

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	サイズ		実効アドレス					
										モード			レジスタ		

サイズ	対応ビット 7	6
バイト	0	0
ワード	0	1
ロングワード	1	0

実効アドレス(データ・可変モード)

アドレッシングモード	対応ビット 5	4	3	2	1	0
Dn	0	0	0	レジスタ番号		
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1

◉アドレッシング・モード

sou	size	dest Dn #	Dn ~	(An) #	(An) ~	(An)+ #	(An)+ ~	−(An) #	−(An) ~	d16(An) #	d16(An) ~	d8(An.IX) #	d8(An.IX) ~	Abs.W #	Abs.W ~	Abs.L #	Abs.L ~
	B	2	4	2	8	2	8	2	10	4	12	4	14	4	12	6	16
	W	2	4	2	8	2	8	2	10	4	12	4	14	4	12	6	16
	L	2	4	2	12	2	12	2	14	4	16	4	18	4	16	6	20

＊ソース・オペランドは存在しない

◉サンプル・リスト

```
TST     D0
TST     (A0)
```

46◉TAS [Test and Set an operand　テスト・アンド・セット]

TAS {.B} 〈ea〉	X	N	Z	V	C
	—	*	*	0	0

解説　マルチプロセッサ・システム（複数のMPUからなるシステム）間の同期をとるための命令であり，オペランドは共有RAMに対するものです．

操作はリード～モディファイ～ライトサイクルで行われ，ハードウエアでこのようなサイクルがサポートされていなければ意味をもたない命令です．

順に次のような操作を行います．

① 指定したバイトオペランドをテスト（ゼロと比較）し，その結果をNとZビットに反映する．

② オペランドの最上位ビット（ビット7）をセット（1）する．

CCR

X：変化せず

N：演算の結果，データの最上位ビット(MSB)が〝1〞ならセット(1)，それ以外はリセット(0)

Z：演算結果がゼロならセット(1)，それ以外はリセット(0)

V：常にリセット(0)

C：常にリセット(0)

◉アドレッシング・モード

sou	size	dest Dn #	Dn ～	(An) #	(An) ～	(An)+ #	(An)+ ～	−(An) #	−(An) ～	d16(An) #	d16(An) ～	d8(An,IX) #	d8(An,IX) ～	Abs.W #	Abs.W ～	Abs.L #	Abs.L ～
	B	2	4	2	14	2	14	2	16	4	18	4	20	4	18	6	22
	W																
	L																

＊ソース・オペランドは存在しない

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	1	1	実効アドレス モード			実効アドレス レジスタ		

実効アドレス（データ・可変モード）

アドレッシングモード	対応ビット 5	4	3	2	1	0
Dn	0	0	0	レジスタ番号		
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1

◉サンプル・リスト

```
TAS     (AØ)
TST     $12FØØ
```

47 ◉AND [AND logical 論理積]

AND {.B/.W/.L} Dn, <ea>	X	N	Z	V	C
	—	*	*	0	0

解説　ディスティネーション・オペランドとソース・オペランドの論理積をとり，結果をディスティネーションへ格納します．

（注） オペランドにアドレスレジスタを指定できません．

CCR
X：変化せず
N：演算の結果，データの最上位ビット（MSB）が"1"
　ならセット(1)，それ以外はリセット(0)
Z：演算結果がゼロならセット(1)，それ以外はリセット(0)
V：常にリセット(0)
C：常にリセット(0)

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	レジスタ			OPモード			実効アドレス					
										モード			レジスタ		

レジスタ → ソース・オペランドのデータレジスタ番号（000～111）

サイズ	対応ビット		
	8	7	6
バイト	1	0	0
ワード	1	0	1
ロングワード	1	1	0

実効アドレス（メモリ・可変モード）

アドレッシングモード	対応ビット					
	5	4	3	2	1	0
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1

◉アドレッシング・モード

sou	size	dest (An) #	(An) ～	(An)+ #	(An)+ ～	−(An) #	−(An) ～	d16(An) #	d16(An) ～	d8(An,IX) #	d8(An,IX) ～	Abs.W #	Abs.W ～	Abs.L #	Abs.L ～
Dn	B	2	12	2	12	2	14	4	16	4	18	4	16	6	20
	W	2	12	2	12	2	14	4	16	4	18	4	16	6	20
	L	2	20	2	20	2	22	4	24	4	26	4	24	6	28

◉サンプル・リスト

```
AND     DØ,(AØ)
AND     DØ,1Ø(A1,A4.L)
```

48◉AND [AND logical 論理積]

AND {.B／.W／.L} 〈ea〉, Dn	X	N	Z	V	C
	−	＊	＊	0	0

解説 ディスティネーション・オペランドとソース・オペランドの論理積をとり，結果をディスティネーションへ格納します．
（注） オペランドにアドレスレジスタを指定できません．

CCR
X：変化せず
N：演算の結果，データの最上位ビット（MSB）が〝1〞ならセット(1)，それ以外はリセット(0)
Z：演算結果がゼロならセット(1)，それ以外はリセット(0)
C：常にリセット(0)
C：常にリセット(0)

◉アドレッシング・モード

sou	size	dest Dn #	dest Dn ～
Dn	B	2	4
	W	2	4
	L	2	8
(An)	B	2	8
	W	2	8
	L	2	14
(An)+	B	2	8
	W	2	8
	L	2	14
−(An)	B	2	10
	W	2	10
	L	2	16
d16(An)	B	4	12
	W	4	12
	L	4	18
d8(An,IX)	B	4	14
	W	4	14
	L	4	20
Abs.W	B	4	12
	W	4	12
	L	4	18
Abs.L	B	6	16
	W	6	16
	L	6	22
d16(PC)	B	4	12
	W	4	12
	L	4	18
d8(PC,IX)	B	4	14
	W	4	14
	L	4	20
＃Imm (注)	B	4	8
	W	4	8
	L	6	14

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	レジスタ			ＯＰモード			実効アドレス					
										モード			レジスタ		

サイズ	対応ビット 8 7 6
バイト	0 0 0
ワード	0 0 1
ロングワード	0 1 0

ディスティネーション・オペランドのデータレジスタ番号(000〜111)

実効アドレス(データモード)

アドレッシングモード	5	4	3	2 1 0
Dn	0	0	0	レジスタ番号
(An)	0	1	0	レジスタ番号
(An)+	0	1	1	レジスタ番号
−(An)	1	0	0	レジスタ番号
d16(An)	1	0	1	レジスタ番号
d8(An,IX)	1	1	0	レジスタ番号
Abs.W	1	1	1	0 0 0
Abs.L	1	1	1	0 0 1
d16(PC)	1	1	1	0 1 0
d8(PC,IX)	1	1	1	0 1 1
(注)＃Imm	1	1	1	1 0 0

（注）ANDIを使用する

◉サンプル・リスト

```
AND     D1,D2
AND     1Ø(A1,A4.L),DØ
```

49 ◉ANDI [AND Immediate イミディエイト論理積]

ANDI {.B/.W/.L} ＃ <data>, <ea>

X	N	Z	V	C
—	*	*	0	0

解説

ディスティネーション・オペランドとイミディエイト値との論理積をとり，結果をディスティネーションへ格納します．

イミディエイト値のサイズは，オペレーションサイズと同じでなければならず，指定したサイズに入りきらない大きさのイミディエイト値は指定できません．

CCR

X：変化せず
N：演算の結果，データの最上位ビット(MSB)が"1"ならセット(1)，それ以外はリセット(0)
Z：演算結果がゼロならセット(1)，それ以外はリセット(0)
V：常にリセット(0)
C：常にリセット(0)

◉アドレッシング・モード

sou	size	dest															
		Dn		(An)		(An)＋		−(An)		d16(An)		d8(An.IX)		Abs.W		Abs.L	
		＃	～	＃	～	＃	～	＃	～	＃	～	＃	～	＃	～	＃	～
＃Imm	B	4	8	4	16	4	16	4	18	6	20	6	22	6	20	8	24
	W	4	8	4	16	4	16	4	18	6	20	6	22	6	20	8	24
	L	6	16	6	28	6	28	6	30	8	32	8	24	8	32	10	36

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	サイズ		実効アドレス					
										モード			レジスタ		

サイズ	対応ビット	
	7	6
バイト	0	0
ワード	0	1
ロングワード	1	0

実効アドレス(データ・可変モード)

アドレッシングモード	対応ビット					
	5	4	3	2	1	0
Dn	0	0	0	レジスタ番号		
(An)	0	1	0	レジスタ番号		
(An)＋	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1

〈第2ワード：サイズがバイトおよびワードの時〉

15 〜 8	7 〜 0

サイズがバイトの時に使用され、指定されたイミディエイト値の8ビットが格納される（ビット7〜0）

サイズがワードの時に使用され、指定されたイミディエイト値の16ビットが格納される（ビット15〜0）

〈第2，第3ワード：サイズがロングワードの時〉

15 〜 0
指定したイミディエイト値の上位ワード
指定したイミディエイト値の下位ワード

◉サンプル・リスト

```
ANDI.W  #$FF00,D0
ANDI.L  #$FF00FF00,(A0)
```

50◉ANDI to CCR [CCRとのイミディエイト論理積]

ANDI {.B} ＃ 〈data〉, CCR	X	N	Z	V	C
	*	*	*	*	*

解説 CCRとイミディエイト値との論理積をとり、結果をCCRへ格納します．
イミディエイト値のサイズは、オペレーションサイズと同じでなければならず、指定したサイズに入りきらない大きさのイミディエイト値は指定できません．
（本命令はCCRの個別ビットをリセットするのに便利である）

CCR
X：ソース・オペランドの対応ビット（ビット4）が〝0〞ならリセット(0)，それ以外は変化せず
N：ソース・オペランドの対応ビット（ビット3）が〝0〞ならリセット(0)，それ以外は変化せず
Z：ソース・オペランドの対応ビット（ビット2）が〝0〞ならリセット(0)，それ以外は変化せず
V：ソース・オペランドの対応ビット（ビット1）が〝0〞ならリセット(0)，それ以外は変化せず
C：ソース・オペランドの対応ビット（ビット0）が〝0〞ならリセット(0)，それ以外は変化せず

◉アドレッシング・モード

sou	size	dest CCR ＃	dest CCR ～
＃Imm	B	4	20
	W		
	L		

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	0	0	1	1	1	1	0	0

〈第2ワード〉

15	14	13	12	11	10	9	8	7–0
0	0	0	0	0	0	0	0	バイトデータ

→指定したイミディエイト値（8ビット）が格納される

◉サンプル・リスト

```
ANDI    #Ø,CCR
```

51 ◉ANDI to SR［特権命令］［SRとのイミディエイト論理積］

ANDI {.W} # <data>, SR	X	N	Z	V	C
	*	*	*	*	*

解説

SRとイミディエイト値との論理積をとり，結果をSRへ格納します．

イミディエイト値のサイズは，オペレーションサイズと同じでなければならず，指定したサイズに入りきらない大きさのイミディエイト値は指定できません．

（本命令はSRの個別ビットをリセットするのに便利である）

SR

T：ソース・オペランドの対応ビット（ビット15，トレース）の値が"0"ならリセット(0)，それ以外は変化せず

S：ソース・オペランドの対応ビット（ビット13，スーパバイザ状態）の値が"0"ならリセット(0)，それ以外は変化せず

I_2：ソース・オペランドの対応ビット（ビット10，割り込みマスク）の値が"0"ならリセット(0)，それ以外は変化せず

I_1：ソース・オペランドの対応ビット（ビット9，割り込みマスク）の値が"0"ならリセット(0)，それ以外は変化せず

I_0：ソース・オペランドの対応ビット（ビット8，割り込みマスク）の値が"0"ならリセット(0)，それ以外は変化せず

X：ソース・オペランドの対応ビット（ビット4）が"0"ならリセット(0)，それ以外は変化せず

N：ソース・オペランドの対応ビット（ビット3）が"0"ならリセット(0)，それ以外は変化せず

Z：ソース・オペランドの対応ビット（ビット2）が"0"ならリセット(0)，それ以外は変化せず

V：ソース・オペランドの対応ビット（ビット1）が"0"ならリセット(0)，それ以外は変化せず

C：ソース・オペランドの対応ビット（ビット0）が"0"ならリセット(0)，それ以外は変化せず

◉アドレッシング・モード

sou	size	dest SR #	dest SR ~
# Imm	B		
	W	4	20
	L		

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	0	1	1	1	1	1	0	0

〈第2ワード〉

15 … 0

ワードデータ

→指定したイミディエイト値(16ビット)が格納される

◉サンプル・リスト

```
ANDI    #$FF00,SR
```

52 ◉EOR [Exclusive OR logical　排他的論理和]

EOR {.B／.W／.L} Dn, 〈ea〉	X	N	Z	V	C
	—	*	*	0	0

解説　ディスティネーション・オペランドとソース・オペランドとの排他的論理和をとり，結果をディスティネーションへ格納しますが，ソース・オペランドにはデータレジスタのみ指定できます．

（注）● EOR 〈ea〉，Dnという形式は存在しません．

● オペランドにアドレスレジスタを指定できません．

CCR

X：変化せず

N：演算の結果，データの最上位ビット（MSB）が〝1〞ならセット(1)，それ以外はリセット(0)

Z：演算結果がゼロならセット(1)，それ以外はリセット(0)

V：常にリセット(0)

C：常にリセット(0)

◉機械語フォーマット

15	14	13	12	11 10 9	8 7 6	5 4 3	2 1 0
1	0	1	1	レジスタ	OPモード	実効アドレス モード	実効アドレス レジスタ

ソース・オペランドで指定したデータレジスタ番号（000～111）

サイズ	対応ビット 8	7	6
バイト	1	0	0
ワード	1	0	1
ロングワード	1	1	0

実効アドレス（データ・可変モード）

アドレッシングモード	対応ビット 5	4	3	2	1	0
Dn	0	0	0	レジスタ番号		
(An)	0	1	0	レジスタ番号		
(An)＋	0	1	1	レジスタ番号		
－(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1

◉アドレッシング・モード

sou	size	Dn #	Dn ～	(An) #	(An) ～	(An)＋ #	(An)＋ ～	－(An) #	－(An) ～	d16(An) #	d16(An) ～	d8(An,IX) #	d8(An,IX) ～	Abs.W #	Abs.W ～	Abs.L #	Abs.L ～
Dn	B	2	4	2	12	2	12	2	14	4	16	4	18	4	16	6	20
	W	2	4	2	12	2	12	2	14	4	16	4	18	4	16	6	20
	L	2	8	2	20	2	20	2	22	4	24	4	26	4	24	6	28

(dest columns: Dn, (An), (An)＋, －(An), d16(An), d8(An,IX), Abs.W, Abs.L)

◉サンプル・リスト

```
EOR      D2,D3
EOR.L    DØ,(AØ)
EOR      D3,$1ØØØ
```

53 ◉EORI [Exclusive OR Immediate　イミディエイト・データとの排他的論理和]

EORI {.B／.W／.L} ＃ 〈data〉, 〈ea〉	X	N	Z	V	C
	―	＊	＊	0	0

解説

ディスティネーション・オペランドとイミディエイト値との排他的論理和をとり，結果をディスティネーションへ格納します．

イミディエイト値のサイズは，オペレーションサイズと同じでなければならず，指定したサイズに入りきらない大きさのイミディエイト値は指定できません．

CCR

X：変化せず
N：演算の結果，データの最上位ビット(MSB)が"1"ならセット(1)，それ以外はリセット(0)
Z：演算結果がゼロならセット(1)，それ以外はリセット(0)
V：常にリセット(0)
C：常にリセット(0)

◉アドレッシング・モード

sou	size	dest															
		Dn		(An)		(An)＋		－(An)		d16(An)		d8(An,IX)		Abs.W		Abs.L	
		＃	～	＃	～	＃	～	＃	～	＃	～	＃	～	＃	～	＃	～
＃Imm	B	4	8	4	16	4	16	4	18	6	20	6	22	6	20	8	24
	W	4	8	4	16	4	16	4	18	6	20	6	22	6	20	8	24
	L	6	16	6	28	6	28	6	30	8	32	8	34	8	32	10	36

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	サイズ		実効アドレス					
										モード			レジスタ		

サイズ	対応ビット	
	7	6
バイト	0	0
ワード	0	1
ロングワード	1	0

実効アドレス(データ・可変モード)

アドレッシングモード	対応ビット					
	5	4	3	2	1	0
Dn	0	0	0	レジスタ番号		
(An)	0	1	0	レジスタ番号		
(An)＋	0	1	1	レジスタ番号		
－(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1

〈第2ワード：サイズがバイトおよびワードの時〉

15 ～ 8	7 ～ 0

サイズがバイトの時に使用され、指定されたイミディエイト値の8ビットが格納される

サイズがワードの時に使用され、指定されたイミディエイト値の16ビットが格納される

〈第2，第3ワード：サイズがロングワードの時〉

15 ～ 0
指定したイミディエイト値の上位ワード
指定したイミディエイト値の下位ワード

◉サンプル・リスト

```
EORI    #$FFFF,DØ
EORI    #$FFØØ,(A2)
```

54◉EORI to CCR [CCRとのイミディエイト排他的論理和]

EORI {.B} # <data>, CCR	X	N	Z	V	C
	*	*	*	*	*

解説　ディスティネーション・オペランド（CCR）とイミディエイト値との排他的論理和をとり，結果をCCRへ格納します．

イミディエイト値のサイズは，オペレーションサイズと同じでなければならず，指定したサイズに入りきらない大きさのイミディエイト値は指定できません．

（本命令はCCRの個別ビットを反転するのに便利である）

CCR

X：ソース・オペランドの対応ビット（ビット4）が〝1〞なら反転，それ以外は変化せず

N：ソース・オペランドの対応ビット（ビット3）が〝1〞なら反転，それ以外は変化せず

Z：ソース・オペランドの対応ビット（ビット2）が〝1〞なら反転，それ以外は変化せず

V：ソース・オペランドの対応ビット（ビット1）が〝1〞なら反転，それ以外は変化せず

C：ソース・オペランドの対応ビット（ビット0）が〝1〞なら反転，それ以外は変化せず

◉アドレッシング・モード

sou	size	dest CCR #	dest CCR ～
# Imm	B	4	20
	W		
	L		

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	0	0	1	1	1	1	0	0

〈第2ワード〉

15	14	13	12	11	10	9	8	7～0
0	0	0	0	0	0	0	0	バイトデータ

→指定したイミディエイト値（8ビット）が格納される

◉サンプル・リスト

```
EORI    #$FF,CCR
```

55 ◉EORI to SR［特権命令］［SRとのイミディエイト排他的論理和］

EORI {.W} ＃ <data>, SR	X	N	Z	V	C
	*	*	*	*	*

解説　ディスティネーション・オペランドとイミディエイト値との排他的論理和をとり，結果をSRへ格納します．
　イミディエイト値のサイズは，オペレーションサイズと同じでなければならず，指定したサイズに入りきらない大きさのイミディエイト値は指定できません．
（本命令はSRの個別ビットを反転するのに便利である）

SR

T：ソース・オペランドの対応ビット（ビット15，トレース）の値が〝1〟なら反転，それ以外は変化せず

S：ソース・オペランドの対応ビット（ビット13，スーパバイザ状態）の値が〝1〟なら反転，それ以外は変化せず

I_2：ソース・オペランドの対応ビット（ビット10，割り込みマスク）の値が〝1〟なら反転，それ以外は変化せず

I_1：ソース・オペランドの対応ビット（ビット9，割り込みマスク）の値が〝1〟なら反転，それ以外は変化せず

I_0：ソース・オペランドの対応ビット（ビット8，割り込みマスク）の値が〝1〟なら反転，それ以外は変化せず

X：ソース・オペランドの対応ビット(ビット4)が〝1〟なら反転，それ以外は変化せず

N：ソース・オペランドの対応ビット（ビット3）が〝1〟なら反転，それ以外は変化せず

Z：ソース・オペランドの対応ビット（ビット2）が〝1〟なら反転，それ以外は変化せず

V：ソース・オペランドの対応ビット（ビット1）が〝1〟なら反転，それ以外は変化せず

C：ソース・オペランドの対応ビット(ビット0)が〝1〟なら反転，それ以外変化せず

◉アドレッシング・モード

sou	size	dest SR ＃	dest SR ～
＃Imm	B		
	W	4	20
	L		

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	0	1	1	1	1	1	0	0

〈第2ワード〉

15 ～ 0
ワードデータ

→指定したイミディエイト値(16ビット)が格納される

◉サンプル・リスト

```
        EORI    #$4,SR
```

56◉OR [Inclusive OR logical 論理和]

OR {.B/.W/.L} Dn, <ea>

X	N	Z	V	C
—	*	*	0	0

解 説

ディスティネーション・オペランドとソース・オペランドとの論理和をとり，結果をディスティネーションへ格納します．

（注） オペランドにアドレスレジスタを指定できません．

CCR

X：変化せず

N：演算の結果，データの最上位ビット（MSB）が〝1〞ならセット(1)，それ以外はリセット(0)

Z：演算結果がゼロならセット(1)，それ以外はリセット(0)

V：常にリセット(0)

C：常にリセット(0)

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	レジスタ			OPモード			実効アドレス モード			実効アドレス レジスタ		

サイズ	対応ビット 8	7	6
バイト	1	0	0
ワード	1	0	1
ロングワード	1	1	0

実効アドレス(メモリ・可変モード)

アドレッシングモード	対応ビット 5	4	3	2	1	0
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1

ソース・オペランドで指定したデータレジスタ番号(000〜111)

◉アドレッシング・モード

sou	size	dest (An) #	(An) 〜	(An)+ #	(An)+ 〜	−(An) #	−(An) 〜	d16(An) #	d16(An) 〜	d8(An,IX) #	d8(An,IX) 〜	Abs.W #	Abs.W 〜	Abs.L #	Abs.L 〜
Dn	B	2	12	2	12	2	14	4	16	4	18	4	16	6	20
	W	2	12	2	12	2	14	4	16	4	18	4	16	6	20
	L	2	20	2	20	2	22	4	24	4	26	4	24	6	28

◉サンプル・リスト

```
OR      D2,(A1)
```

57 ◉OR [Inclusive OR logical 論理和]

OR {.B/.W/.L} <ea>, Dn	X	N	Z	V	C
	―	*	*	0	0

解説 ディスティネーション・オペランドとソース・オペランドとの論理和をとり，結果をディスティネーションへ格納します．
（注） オペランドにアドレスレジスタを指定できません．

CCR
X：変化せず
N：演算の結果，データの最上位ビット（MSB）が"1"ならセット(1)，それ以外はリセット(0)
Z：演算結果がゼロならセット(1)，それ以外はリセット(0)
V：常にリセット(0)
C：常にリセット(0)

◉アドレッシング・モード

sou	size	dest Dn #	dest Dn ~
Dn	B	2	4
	W	2	4
	L	2	8
(An)	B	2	8
	W	2	8
	L	2	14
(An)+	B	2	8
	W	2	8
	L	2	14
−(An)	B	2	10
	W	2	10
	L	2	16
d16(An)	B	4	12
	W	4	12
	L	4	18
d8(An,IX)	B	4	14
	W	4	14
	L	4	20
Abs.W	B	4	12
	W	4	12
	L	4	18
Abs.L	B	6	16
	W	6	16
	L	6	22
d16(PC)	B	4	12
	W	4	12
	L	4	18
d8(PC,IX)	B	4	14
	W	4	14
	L	4	20
# Imm (注)	B	4	8
	W	4	8
	L	6	14

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	レジスタ			OPモード			実効アドレス モード			実効アドレス レジスタ		

サイズ	対応ビット 8	7	6
バイト	0	0	0
ワード	0	0	1
ロングワード	0	1	0

ディスティネーション・オペランドで指定したデータレジスタ番号（000～111）

実効アドレス（データモード）

アドレッシングモード	5	4	3	2	1	0
Dn	0	0	0	レジスタ番号		
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1
d16(PC)	1	1	1	0	1	0
d8(PC,IX)	1	1	1	0	1	1
(注)# Imm	1	1	1	1	0	0

（注）ORIを使用する

◉サンプル・リスト

```
OR      D2,D4
```

58 ◉ORI [Inclusive OR Immediate　イミディエイト論理和]

ORI {.B／.W／.L} ＃ 〈data〉, 〈ea〉	X	N	Z	V	C
	—	*	*	0	0

解説　ディスティネーション・オペランドとソース・オペランドとの論理和をとり，結果をディスティネーションへ格納します．

イミディエイト値のサイズは，オペレーションサイズと同じでなければならず，指定したサイズに入りきらない大きさのイミディエイト値は指定できません．

CCR

X：変化せず

N：演算の結果，データの最上位ビット（MSB）が〝1″ならセット(1)，それ以外はリセット(0)

Z：演算結果がゼロならセット(1)，それ以外はリセット(0)

V：常にリセット(0)

C：常にリセット(0)

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	サイズ		実効アドレス					
										モード			レジスタ		

サイズ	対応ビット	
	7	6
バイト	0	0
ワード	0	1
ロングワード	1	0

実効アドレス(データ・可変モード)

アドレッシングモード	対応ビット					
	5	4	3	2	1	0
Dn	0	0	0	レジスタ番号		
(An)	0	1	0	レジスタ番号		
(An)＋	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An, IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1

〈第2ワード：サイズがバイトおよびワードの時〉

15　　　　8	7　　　　0

→ サイズがバイトの時に使用され，指定されたイミディエイト値の8ビットが格納される（ビット7〜0）

→ サイズがワードの時に使用され，指定されたイミディエイト値の16ビットが格納される（ビット15〜0）

〈第2，第3ワード：サイズがロングワードの時〉

15　　　　0
指定したイミディエイト値の上位ワード
指定したイミディエイト値の下位ワード

◉アドレッシング・モード

sou	size	dest															
		Dn		(An)		(An)＋		−(An)		d16(An)		d8(An,IX)		Abs.W		Abs.L	
		＃	〜	＃	〜	＃	〜	＃	〜	＃	〜	＃	〜	＃	〜	＃	〜
＃Imm	B	4	8	4	16	4	16	4	18	6	20	6	22	6	20	8	24
	W	4	8	4	16	4	16	4	18	6	20	6	22	6	20	8	24
	L	6	16	6	30	6	30	6	32	8	34	8	36	8	34	10	38

◉サンプル・リスト

```
ORI     #$FFFF,DØ
ORI     #$FFØØ,4(A1,A2.L)
```

59 ◉ORI to CCR [CCRとのイミディエイト論理和]

ORI {.B} ＃ ⟨data⟩, CCR

X	N	Z	V	C
*	*	*	*	*

解説　ディスティネーション・オペランド（CCR）とソース・オペランドとの論理和をとり，結果をCCRへ格納します．

イミディエイト値のサイズは，オペレーションサイズと同じでなければならず，指定したサイズに入りきらない大きさのイミディエイト値は指定できません．

（本命令はCCRの個別ビットをセットするのに便利である）

CCR

X：ソース・オペランドの対応ビット（ビット4）が〝1″ならセット(1)，それ以外は変化せず

N：ソース・オペランドの対応ビット（ビット3）が〝1″ならセット(1)，それ以外は変化せず

Z：ソース・オペランドの対応ビット（ビット2）が〝1″ならセット(1)，それ以外は変化せず

V：ソース・オペレンドの対応ビット（ビット1）が〝1″ならセット(1)，それ以外は変化せず

C：ソース・オペランドの対応ビット（ビット0）が〝1″ならセット(1)，それ以外は変化せず

◉アドレッシング・モード

sou	size	dest CCR ＃	dest CCR ～
＃Imm	B	4	20
	W		
	L		

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0

〈第2ワード〉

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	バイトデータ							

→指定したイミディエイト値（8ビット）が格納される

◉サンプル・リスト

```
ORI     #$FF,CCR
```

60◉ORI to SR［特権命令］［SRとのイミディエイト論理和］

ORI {.W} ＃ <data>, SR	X	N	Z	V	C
	*	*	*	*	*

解説　ディスティネーション・オペランドとソース・オペランドとの論理和をとり，結果をSRへ格納します．

イミディエイト値のサイズは，オペレーションサイズと同じでなければならず，指定したサイズに入りきらない大きさのイミディエイト値は指定できません．

（本命令はSRの個別ビットをセットするのに便利である）

SR

T：ソース・オペランドの対応ビット（ビット15，トレース）の値が〝1〟ならセット（1），それ以外は変化せず

S：ソース・オペランドの対応ビット（ビット13，スーパバイザ状態）の値が〝1〟ならセット（1），それ以外は変化せず

I_2：ソース・オペランドの対応ビット（ビット10，割り込みマスク）の値が〝1〟ならセット（1），それ以外は変化せず

I_1：ソース・オペランドの対応ビット（ビット9，割り込みマスク）の値が〝1〟ならセット（1），それ以外は変化せず

I_0：ソース・オペランドの対応ビット（ビット8，割り込みマスク）の値が〝1〟ならセット（1），それ以外は変化せず

X：ソース・オペランドの対応ビット（ビット4）が〝1〟ならセット（1），それ以外は変化せず

N：ソース・オペランドの対応ビット（ビット3）が〝1〟ならセット（1），それ以外は変化せず

Z：ソース・オペランドの対応ビット（ビット2）が〝1〟ならセット（1），それ以外は変化せず

V：ソース・オペランドの対応ビット（ビット1）が〝1〟ならセット（1），それ以外は変化せず

C：ソース・オペランドの対応ビット（ビット0）が〝1〟ならセット（1），それ以外は変化せず

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0

〈第2ワード〉

15 … 0
ワードデータ

→指定したイミディエイト値(16ビット)が格納される

◉アドレッシング・モード

sou	size	dest SR ＃	dest SR ～
＃Imm	B		
	W	4	20
	L		

◉サンプル・リスト

```
        ORI     #$FF,SR
```

61 ●NOT [Logical Complement 論理否定]

NOT {.B/.W/.L} 〈ea〉

X	N	Z	V	C
—	*	*	0	0

解説 ディスティネーション・オペランドの1の補数をとり，結果をディスティネーションへ格納します．

CCR
X：変化せず
N：演算結果が負ならセット（1)、それ以外はリセット（0）
Z：演算結果がゼロならセット（1)、それ以外はリセット（0）
V：常にリセット（0）
C：常にリセット（0）

●機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	0	サイズ		実効アドレス					
										モード			レジスタ		

サイズ	対応ビット	
	7	6
バイト	0	0
ワード	0	1
ロングワード	1	0

実効アドレス（データ・可変モード）

アドレッシングモード	対応ビット			
	5	4	3	2 1 0
Dn	0	0	0	レジスタ番号
(An)	0	1	0	レジスタ番号
(An)+	0	1	1	レジスタ番号
−(An)	1	0	0	レジスタ番号
d16(An)	1	0	1	レジスタ番号
d8(An,IX)	1	1	0	レジスタ番号
Abs.W	1	1	1	0 0 0
Abs.L	1	1	1	0 0 1

●アドレッシング・モード

sou	size	dest															
		Dn		(An)		(An)+		−(An)		d16(An)		d8(An,IX)		Abs.W		Abs.L	
		#	~	#	~	#	~	#	~	#	~	#	~	#	~	#	~
	B	2	4	2	12	2	12	2	14	4	16	4	18	4	16	6	20
	W	2	4	2	12	2	12	2	14	4	16	4	18	4	16	6	20
	L	2	6	2	20	2	20	2	22	4	24	4	26	4	24	6	28

*ソース・オペランドは存在しない

●サンプル・リスト

```
NOT     D0
NOT     (A0)
```

62 ◉ASL [Arithmetic Shift Left 左へ算術シフト]

ASL {.B/.W/.L} Dn, Dn	X	N	Z	V	C
	*	*	*	*	*

解説　ディスティネーション・オペランドを，ソース・オペランドのデータレジスタの内容だけ，左へ算術シフトしますが，シフトカウンタに使用されるソース・オペランドのデータレジスタ値は，64の余りが用いられ，0～63までの範囲となります．

シフトの様子を次に示します。

Dn

C ← ディスティネーションオペランド ← 0

X

●オペランドを左へシフト．
●最上位ビットからシフトされ押し出されたビットは，CおよびXビットへコピーされる．
●下位ビットには0が入る．
●シフト操作によって符号変化が生じた場合，Vビットがセットされる．

CCR

X：Cビットと同じ値であるが，シフトカウント値がゼロなら変化せず
N：演算の結果，データの最上位ビット（MSB）が〝1〟ならセット（1），それ以外はリセット（0）
Z：演算結果がゼロならセット（1），それ以外はリセット（0）
V：シフト・オペレーション中一度でもデータの最上位ビット（MSB）が変化すればセット（1），それ以外はリセット（0）
C：最後にシフトされて押し出されたビット値を保持，ただし，シフトカウント値がゼロなら常にリセット（0）

◉アドレッシング・モード

sou	size	dest Dn #	dest Dn ~
Dn	B	2	6+2n
	W	2	6+2n
	L	2	8+2n

◉機械語フォーマット

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	1	1	0	カウントまたはレジスタ	dr	サイズ	i/r	0	0	レジスタ

- レジスタ → ディスティネーション・レジスタ番号(000～111)（操作されるデータレジスタの番号）
- i/r → 1：シフト回数はソース・オペランドのデータレジスタで指定する
- サイズ → 00：バイト操作　01：ワード操作　10：ロングワード操作
- dr → 1：左シフト
- カウントまたはレジスタ → ソースレジスタ番号(000～111)（シフト回数が入っているデータレジスタ番号）

◉サンプル・リスト

```
ASL      D2,D7
```

63◉ASL [Arithmetic Shift Left 左へ算術シフト]

ASL {.B/.W/.L} #<data>, Dn

X	N	Z	V	C
*	*	*	*	*

解説 ディスティネーション・オペランドを，ソース・オペランドのイミディエイト値だけ，左へ算術シフトしますが，シフトカウントとして指定できる範囲は1～8です．

シフトの様子を次に示します．

- オペランドを左へシフト．
- シフトカウントは1～8．
- 最上位ビットからシフトされ押し出されたビットは，CおよびXビットへコピーされる．
- 下位ビットには0が入る．
- シフト操作によって符号変化が生じた場合，Vビットがセットされる．

CCR

X：Cビットと同じ値

N：演算の結果，データの最上位ビット（MSB）が"1"ならセット（1），それ以外はリセット（0）

Z：演算結果がゼロならセット（1），それ以外はリセット（0）

V：シフト・オペレーション中一度でもデータの最上位ビット（MSB）が変化すればセット（1），それ以外はリセット（0）

C：最後にシフトされて押し出されたビット値を保持

◉アドレッシング・モード

sou	size	dest Dn #	dest Dn ～
#Imm	B	2	6+2n
	W	2	6+2n
	L	2	8+2n

◉機械語フォーマット

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	1	1	0	カウントまたはレジスタ	dr	サイズ	i/r	0	0	レジスタ

- レジスタ → ディスティネーション・レジスタ番号(000～111)（操作されるデータレジスタの番号）
- i/r → 0：シフト回数は，ソース・オペランドのイミディエイト値で指定する
- サイズ → 00：バイト操作 / 01：ワード操作 / 10：ロングワード操作
- dr → 1：左シフト
- カウント → シフト回数(1～8，ただし0は8を表す)

◉サンプル・リスト

```
ASL     #4,D0
```

64●ASL [Arithmetic Shift Left 左へ算術シフト]

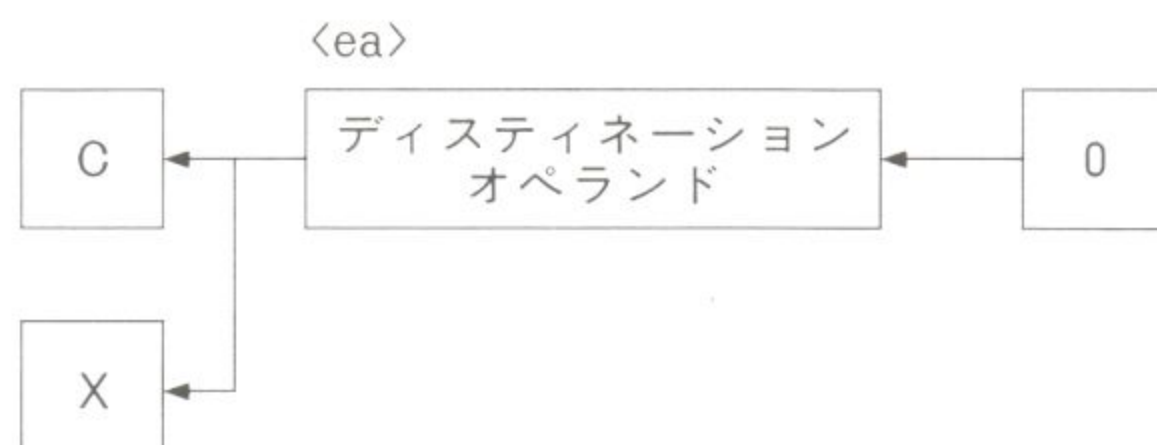
ASL {.W} <ea>	X	N	Z	V	C
	*	*	*	*	*

解説　ディスティネーションで指定したメモリ・オペランドを，1ビットだけ，左へ算術シフトしますが，シフト回数(1回)やオペレーションサイズ(ワード)が固定されています．

シフトの様子を次に示します（シフト回数は固定されている）．

<ea>

C ← ディスティネーションオペランド ← 0

X ←

- オペランドを左へシフト．
- シフトカウンタは1に固定されている．
- 最上位ビットからシフトされ押し出されたビットは，CおよびXビットへコピーされる．
- 下位ビットには0が入る．
- シフト操作によって符号変化が生じた場合，Vビットがセットされる．

CCR

X：Cビットと同じ値

N：演算の結果，データの最上位ビット（MSB）が"1"ならセット（1），それ以外はリセット（0）

Z：演算結果がゼロならセット（1），それ以外はリセット（0）

V：シフト・オペレーション中一度でもデータの最上位ビット（MSB）が変化すればセット（1），それ以外はリセット（0）

C：最後にシフトされて押し出されたビット値を保持

●アドレッシング・モード

sou	size	dest (An) #	(An) ~	(An)+ #	(An)+ ~	−(An) #	−(An) ~	d16(An) #	d16(An) ~	d8(An,IX) #	d8(An,IX) ~	Abs.W #	Abs.W ~	Abs.L #	Abs.L ~
	B														
	W	2	12	2	12	2	14	4	16	4	18	4	16	6	20
	L														

＊ソースオペランドは存在しない

●機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	0	dr	1	1	実効アドレス モード			実効アドレス レジスタ		

dr → 1：左シフト

実効アドレス(メモリ・可変モード)

アドレッシングモード	対応ビット 5	4	3	2	1	0
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1

●サンプル・リスト

```
ASL      (AØ)+
```

65●ASR [Arithmetic Shift Right 右へ算術シフト]

ASR {.B/.W/.L} Dn, Dn	X	N	Z	V	C
	*	*	*	0	*

解説　ディスティネーション・オペランドを，ソース・オペランドのデータレジスタの内容だけ，右へ算術シフトしますが，シフトカウンタに使用されるソース・オペランドのデータレジスタ値は，64の余りが用いられ，0～63までの範囲となります．

シフトの様子を次に示します．

- オペランドを右へシフト．
- 最下位ビットからシフトされ押し出されたビットは，CおよびXビットへコピーされる．
- 符号ビットは最上位ビットへ再度入る．

CCR
X：Cビットと同じ値であるが，シフトカウント値がゼロなら変化せず
N：演算の結果，データの最上位ビット（MSB）が〝1〟ならセット（1），それ以外はリセット（0）
Z：演算結果がゼロならセット（1），それ以外はリセット（0）
V：常にリセット（0）・・・・最上位ビットは変化しない
C：最後にシフトされて押し出されたビット値を保持，ただし，シフトカウント値がゼロなら常にリセット（0）

●アドレッシング・モード

sou	size	dest Dn #	dest Dn ～
Dn	B	2	6+2n
	W	2	6+2n
	L	2	8+2n

●機械語フォーマット

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	1	1	0	カウントまたはレジスタ	dr	サイズ	i/r	0	0	レジスタ

- レジスタ → ディスティネーション・レジスタ番号(000～111)（操作されるデータレジスタの番号）
- i/r → 1：シフト回数はソース・オペランドのデータレジスタで指定する
- サイズ → 00：バイト操作　01：ワード操作　10：ロングワード操作
- dr → 0：右シフト
- カウントまたはレジスタ → ソースレジスタ番号(000～111)（シフト回数が入っているデータレジスタ番号）

●サンプル・リスト

```
ASR     D2,D4
```

66●ASR [Arithmetic Shift Right 右へ算術シフト]

ASR {.B/.W/.L} #〈data〉, Dn	X	N	Z	V	C
	*	*	*	0	*

解説 ディスティネーション・オペランドを，ソース・オペランドのイミディエイト値だけ，右へ算術シフトしますが，シフトカウントとして指定できる範囲は1～8です．

シフトの様子を次に示します．

Dn

ディスティネーション オペランド → C, X

- オペランドを右へシフト．
- シフトカウントは1～8．
- 最下位ビットからシフトされ押し出されたビットは，CおよびXビットへコピーされる．
- 符号ビットは最上位ビットへ再度入る．

CCR

X：Cビットと同じ値

N：演算の結果，データの最上位ビット（MSB）が〝1″ならセット（1），それ以外はリセット（0）

Z：演算結果がゼロならセット（1），それ以外はリセット（0）

V：常にリセット（0）‥‥最上位ビットは変化しない

C：最後にシフトされて押し出されたビット値を保持

◉アドレッシング・モード

sou	size	dest Dn #	dest Dn ～
#Imm	B	2	6+2n
	W	2	6+2n
	L	2	8+2n

◉機械語フォーマット

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	1	1	0	カウントまたはレジスタ	dr	サイズ	i/r	0	0	レジスタ

- レジスタ → ディスティネーション・レジスタ番号（操作されるデータレジスタの番号）
- i/r → 0：シフト回数はソース・オペランドのイミディエイト値で指定する
- サイズ → 00：バイト操作 / 01：ワード操作 / 10：ロングワード操作
- dr → 0：右シフト
- カウントまたはレジスタ → シフト回数（1～8，ただし0は8を表す）

◉サンプル・リスト

```
ASR     #4,DØ
```

第3部 命令の詳細

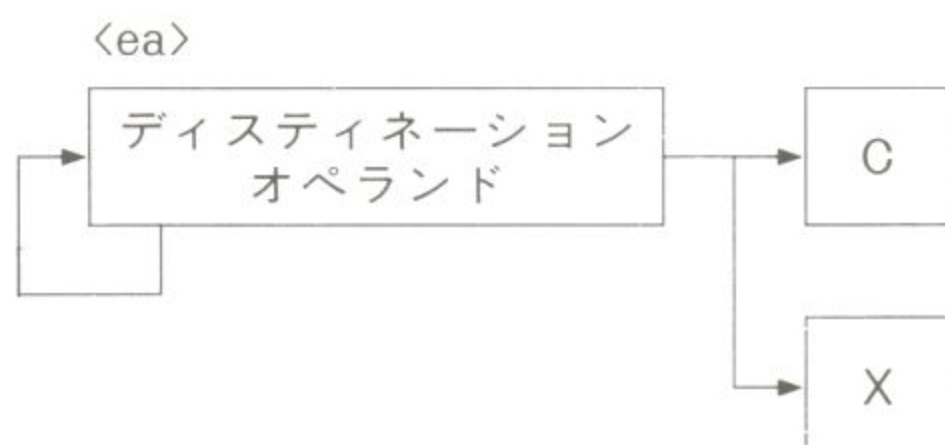
67 ◉ASR [Arithmetic Shift Right 右へ算術シフト]

ASR {.W} 〈ea〉	X	N	Z	V	C
	*	*	*	0	*

解説　ディスティネーションで指定したメモリ・オペランドを，1ビットだけ，右へ算術シフトしますが，シフト回数(1回)やオペレーションサイズ(ワード)が固定されています．

シフトの様子を次に示します．

〈ea〉

ディスティネーション オペランド	→	C
	→	X

- オペランドを右へシフト．
- シフトカウントは1に固定されている．
- 最下位ビットからシフトされ押し出されたビットは，CおよびXビットへコピーされる．
- 符号ビットは最下位ビットへ再度入る．

CCR

X：Cビットと同じ値

N：演算の結果，データの最上位ビット(MSB)が"1"ならセット(1)，それ以外はリセット(0)

Z：演算結果がゼロならセット(1)，それ以外はリセット(0)

V：常にリセット(0)‥‥最上位ビットは変化しない

C：最後にシフトされて押し出されたビット値を保持

◉アドレッシング・モード

sou	size	dest (An) #	(An) ~	(An)+ #	(An)+ ~	−(An) #	−(An) ~	d16(An) #	d16(An) ~	d8(An,IX) #	d8(An,IX) ~	Abs.W #	Abs.W ~	Abs.L #	Abs.L ~
	B														
	W	2	12	2	12	2	14	4	16	4	18	4	16	6	20
	L														

＊ソース・オペランドは存在しない

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	0	dr	1	1	実効アドレス モード			実効アドレス レジスタ		

dr → 0：右シフト

実効アドレス(メモリ・可変モード)

アドレッシングモード	対応ビット 5	4	3	2	1	0
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1

◉サンプル・リスト

```
ASR     (AØ)
ASR     2Ø(A2,DØ.L)
```

68 ●LSL [Logical Shift Left 左へ論理シフト]

LSL {.B/.W/.L} Dn, Dn	X	N	Z	V	C
	*	*	*	0	*

解説　ディスティネーション・オペランドを，ソース・オペランドのデータレジスタの内容だけ，左へ論理シフトしますが，シフトカウンタに使用されるソース・オペランドのデータレジスタ値は，64の余りが用いられ，0～63までの範囲となります．

シフトの様子を次に示します．

- オペランドを左へシフト．
- 最上位ビットからシフトされ押し出されたビットは，CおよびXビットへコピーされる．
- 下位ビットには0が入る．
- Vビットは常にクリア(0)される．

CCR

X：Cビットと同じ値であるが，シフトカウント値がゼロなら変化せず

N：演算の結果，データの最上位ビット（MSB）が"1"ならセット（1），それ以外はリセット（0）

Z：演算結果がゼロならセット(1)，それ以外はリセット（0）

V：常にリセット（0）

C：最後にシフトされて押し出されたビット値を保持，ただし，シフトカウント値がゼロなら常にリセット（0）

●アドレッシング・モード

sou	size	dest Dn #	～
Dn	B	2	6+2n
	W	2	6+2n
	L	2	8+2n

●機械語フォーマット

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	1	1	0	カウントまたはレジスタ	dr	サイズ	i/r	0	1	レジスタ

- レジスタ → ディスティネーション・レジスタ番号(000～111)（操作されるデータレジスタの番号）
- i/r → 1：シフト回数はソース・オペランドのデータレジスタで指定する
- サイズ → 00：バイト操作／01：ワード操作／10：ロングワード操作
- dr → 1：左シフト
- カウントまたはレジスタ → ソースレジスタ番号(000～111)（シフト回数が入っているデータレジスタ番号）

●サンプル・リスト

```
LSL     D1,D2
```

69 ●LSL [Logical Shift Left 左へ論理シフト]

LSL {.B/.W/.L} #〈data〉, Dn	X	N	Z	V	C
	*	*	*	0	*

解説 ディスティネーション・オペランドを，ソース・オペランドのイミディエイト値だけ，左へ論理シフトしますが，シフトカウントとして指定できる範囲は1～8です．

シフトの様子を次に示します．

Dn: C ← ディスティネーション オペランド ← 0; X ← (C)

- ●オペランドを左へシフト．
- ●シフトカウントは1～8．
- ●最上位ビットからシフトされ押し出されたビットは，CおよびXビットへコピーされる．
- ●下位ビットには0が入る．
- ●Vビットは常にクリア(0)される．

CCR

X：Cビットと同じ値

N：演算の結果，データの最上位ビット（MSB）が〝1〞ならセット（1），それ以外はリセット（0）

Z：演算結果がゼロならセット(1)，それ以外はリセット（0）

V：常にリセット（0）

C：最後にシフトされて押し出されたビット値を保持

●アドレッシング・モード

sou	size	dest Dn #	dest Dn ～
#Imm	B	2	6+2n
	W	2	6+2n
	L	2	8+2n

●機械語フォーマット

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	1	1	0	カウントまたはレジスタ	dr	サイズ	i/r	0	1	レジスタ

- レジスタ → ディスティネーション・レジスタ番号(000～111)(操作されるデータレジスタの番号)
- i/r → 0：シフト回数はソース・オペランドのイミディエイト値で指定する
- サイズ → 00：バイト操作 / 01：ワード操作 / 10：ロングワード操作
- dr → 1：左シフト
- カウントまたはレジスタ → シフト回数(1～8，ただし0は8を表す)

●サンプル・リスト

```
LSL     #4,D0
```

70 ●LSL [Logical Shift Left 左へ論理シフト]

LSL {.W} <ea>

X	N	Z	V	C
*	*	*	0	*

解説

ディスティネーション・オペランドを，1ビットだけ左へ論理シフトしますが，シフト回数(1回)やオペレーションサイズ(ワード)が固定されています．

シフトの様子を次に示します．

<ea>

C ← ディスティネーションオペランド ← 0
X ←

- ●オペランドを左へシフト．
- ●シフトカウントは1に固定されている．
- ●最上位ビットからシフトされ押し出されたビットは，CおよびXビットへコピーされる．
- ●下位ビットには0が入る．
- ●Vビットは常にクリア(0)される．

CCR

X：Cビットと同じ値

N：演算の結果，データの最上位ビット（MSB）が〝1〟ならセット(1)，それ以外はリセット（0）

Z：演算結果がゼロならセット（1），それ以外はリセット（0）

V：常にリセット（0）

C：最後にシフトされて押し出されたビット値を保持

●アドレッシング・モード

sou	size	dest (An) #	(An) ~	(An)+ #	(An)+ ~	−(An) #	−(An) ~	d16(An) #	d16(An) ~	d8(An,IX) #	d8(An,IX) ~	Abs.W #	Abs.W ~	Abs.L #	Abs.L ~
	B														
	W	2	12	2	12	2	14	4	16	4	18	4	16	6	20
	L														

＊ソース・オペランドは存在しない

●機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	1	dr	1	1	実効アドレス モード			レジスタ		

dr → 1：左シフト

実効アドレス(メモリ・可変モード)

アドレッシングモード	5	4	3	2	1	0
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1

●サンプル・リスト

```
LSL     (AØ)
LSL     12(A1,A2.L)
```

71 ◉LSR [Logical Shift Right 右へ論理シフト]

LSR {.B/.W/.L} Dn, Dn	X	N	Z	V	C
	*	*	*	0	*

解説 ディスティネーション・オペランドを，ソース・オペランドのデータレジスタの内容だけ，右へ論理シフトしますが，シフトカウンタに使用されるソース・オペランドのデータレジスタ値は，64の余りが用いられ，0～63までの範囲となります．

シフトの様子を次に示します．

- オペランドを右へシフト．
- 最下位ビットからシフトされ押し出されたビットは，CおよびXビットへコピーされる．
- 上位ビットには0が入る．
- Vビットは常にクリア(0)される．

CCR

X：Cビットと同じ値であるが，シフトカウント値がゼロなら変化せず

N：演算の結果，データの最上位ビット（MSB）が〝1〞ならセット（1），それ以外はリセット（0）

Z：演算結果がゼロならセット(1)，それ以外はリセット（0）

V：常にリセット(0)

C：最後にシフトされて押し出されたビット値を保持，ただし，シフトカウント値がゼロなら常にリセット（0）

◉アドレッシング・モード

sou	size	dest Dn #	dest Dn ~
Dn	B	2	6+2n
	W	2	6+2n
	L	2	8+2n

◉機械語フォーマット

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	1	1	0	カウントまたはレジスタ	dr	サイズ	i/r	0	1	レジスタ

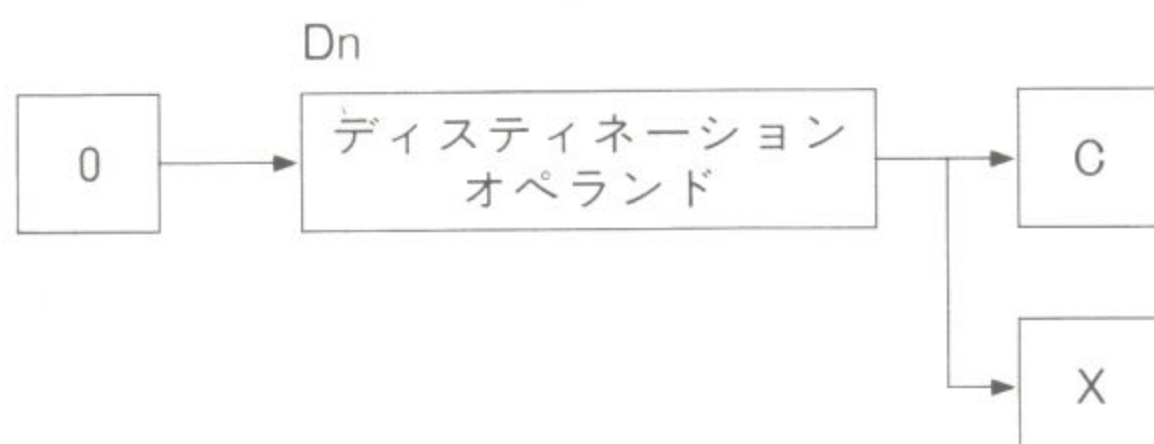
- レジスタ → ディスティネーション・レジスタ番号(000～111)（操作されるデータレジスタの番号）
- i/r → 1：シフト回数はソース・オペランドのデータレジスタで指定する
- サイズ → 00：バイト操作　01：ワード操作　10：ロングワード操作
- dr → 0：右シフト
- カウントまたはレジスタ → ソースレジスタ番号(000～111)（シフト回数が入っているデータレジスタ番号）

◉サンプル・リスト

```
LSR      D2,D4
```

72 ●LSR [Logical Shift Right 右へ論理シフト]

LSR {.B/.W/.L} #<data>, Dn	X	N	Z	V	C
	*	*	*	0	*

解説 ディスティネーション・オペランドを，ソース・オペランドのイミディエイト値だけ，右へ論理シフトしますが，シフトカウントとして指定できる範囲は1～8です．

シフトの様子を次に示します．

Dn

0 → ディスティネーションオペランド → C, X

- ●オペランドを右へシフト．
- ●シフトカウントは1～8．
- ●最下位ビットからシフトされ押し出されたビットは，CおよびXビットへコピーされる．
- ●上位ビットには0が入る．
- ●Vビットは常にクリア(0)される．

CCR

X：Cビットと同じ値

N：演算の結果，データの最上位ビット（MSB）が〝1″ならセット（1），それ以外はリセット（0）

Z：演算結果がゼロならセット(1)，それ以外はリセット（0）

V：常にリセット（0）

C：最後にシフトされて押し出されたビット値を保持

◉アドレッシング・モード

sou	size	dest Dn #	～
# Imm	B	2	6+2n
	W	2	6+2n
	L	2	8+2n

◉機械語フォーマット

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	1	1	0	カウントまたはレジスタ	dr	サイズ	i/r	0	1	レジスタ

- レジスタ → ディスティネーション・レジスタ番号(000～111)(操作されるデータレジスタの番号)
- i/r → 0：シフト回数はソース・オペランドのイミディエイト値で指定する
- サイズ → 00：バイト操作　01：ワード操作　10：ロングワード操作
- dr → 0：右シフト
- カウントまたはレジスタ → シフト回数(1～8，ただし0は8を表す)

◉サンプル・リスト

```
LSR     #4,D0
```

73 ●LSR [Logical Shift Right 右へ論理シフト]

LSR {.W} 〈ea〉	X	N	Z	V	C
	*	*	*	0	*

解説

ディスティネーション・オペランドを，1ビットだけ右へ論理シフトしますが，シフト回転（1回）やオペレーションサイズ（ワード）が固定されています．

シフトの様子を次に示します。

0 → 〈ea〉 ディスティネーションオペランド → C, X

- オペランドを右へシフト．
- シフトカウントは1に固定されている．
- 最下位ビットからシフトされ押し出されたビットは，CおよびXビットへコピーされる．
- 上位ビットには0が入る．
- Vビットは常にクリア（0）される．

CCR

X：Cビットと同じ値

N：演算の結果，データの最上位ビット（MSB）が〝1〞ならセット（1），それ以外はリセット（0）

Z：演算結果がゼロならセット（1），それ以外はリセット（0）

V：常にリセット（0）

C：最後にシフトされて押し出されたビット値を保持

◉アドレッシング・モード

sou	size	dest (An) #	(An) ~	(An)+ #	(An)+ ~	−(An) #	−(An) ~	d16(An) #	d16(An) ~	d8(An,IX) #	d8(An,IX) ~	Abs.W #	Abs.W ~	Abs.L #	Abs.L ~
	B														
	W	2	12	2	12	2	14	4	16	4	18	4	16	6	20
	L														

＊ソース・オペランドは存在しない

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	1	dr	1	1	実効アドレス モード			実効アドレス レジスタ		

dr → 0：右シフト

実効アドレス（メモリ・可変モード）

アドレッシングモード	5	4	3	2 1 0
(An)	0	1	0	レジスタ番号
(An)+	0	1	1	レジスタ番号
−(An)	1	0	0	レジスタ番号
d16(An)	1	0	1	レジスタ番号
d8(An,IX)	1	1	0	レジスタ番号
Abs.W	1	1	1	0 0 0
Abs.L	1	1	1	0 0 1

◉サンプル・リスト

```
LSR     (A0)+
```

74 ◉ROL [Rotate　Left　左へローテート]

ROL　{.B/.W/.L}　Dn, Dn	X	N	Z	V	C
	—	*	*	0	*

解説　ディスティネーション・オペランドを，ソース・オペランドのデータレジスタの内容だけ，左へローテート(回転)しますが，ローテートカウンタに使用されるソース・オペランドのデータレジスタ値は，64の余りが用いられ，0～63までの範囲となります．

ローテートの様子を次に示します．

- オペランドを左へローテート．
- 最上位ビットからローテートされ押し出されたビットは，Cビットだけでなく最下位ビットにも入る．
- Vビットは常にクリア（0）される．

CCR
X：変化せず
N：演算の結果，データの最上位ビット（MSB）が〝1〃ならセット（1），それ以外はリセット（0）
Z：演算結果がゼロならセット(1)，それ以外はリセット（0）
V：常にリセット（0）
C：最後にローテートされて押し出されたビット値を保持，ただし，ローテート・カウント値がゼロなら常にリセット（0）

◉アドレッシング・モード

sou	size	dest Dn #	dest Dn ~
Dn	B	2	6+2n
	W	2	6+2n
	L	2	8+2n

◉機械語フォーマット

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	1	1	0	カウントまたはレジスタ	dr	サイズ	i/r	1	1	レジスタ

- レジスタ → ディスティネーション・レジスタ番号(000～111)(操作されるデータレジスタの番号)
- i/r → 1：ローテート回数はソース・オペランドのデータレジスタで指定する
- サイズ → 00：バイト操作　01：ワード操作　10：ロングワード操作
- dr → 1：左ローテート
- カウントまたはレジスタ → ソースレジスタ番号(000～111)（ローテート回数が入っているデータレジスタ番号）

◉サンプル・リスト

```
ROL      D2,D3
```

75◉ROL [Rotate Left 左へローテート]

ROL {.B/.W/.L} #<data>, Dn

X	N	Z	V	C
—	*	*	0	*

解説 ディスティネーション・オペランドを，ソース・オペランドのイミディエイト値だけ，左へローテート(回転)しますが，ローテートカウントとして指定できる範囲は1～8です．

ローテートの様子を次に示します．

- オペランドを左へローテート．
- ローテートカウントは1～8．
- 最上位ビットからローテートされ押し出されたビットは，Cビットだけでなく最下位ビットにも入る．
- Vビットは常にクリア(0)される．

CCR

X：変化せず
N：演算の結果，データの最上位ビット（MSB）が〝1〞ならセット（1），それ以外はリセット（0）
Z：演算結果がゼロならセット(1)，それ以外はリセット（0）
V：常にリセット（0）
C：最後にローテートされて押し出されたビット値を保持

◉アドレッシング・モード

sou	size	dest Dn #	dest Dn ～
#Imm	B	2	6+2n
	W	2	6+2n
	L	2	8+2n

◉機械語フォーマット

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	1	1	0	カウントまたはレジスタ	dr	サイズ	i/r	1	1	レジスタ

- レジスタ → ディスティネーション・レジスタ番号(000～111)（操作されるデータレジスタの番号）
- i/r → 0：ローテート回数はソース・オペランドのイミディエイト値で指定する
- サイズ → 00：バイト操作　01：ワード操作　10：ロングワード操作
- dr → 1：左ローテート
- カウントまたはレジスタ → シフト回数(1～8，ただし0は8を表す)

◉サンプル・リスト

```
ROL     #5,D0
```

76◉ROL [Rotate Left 左へローテート]

ROL {.W} 〈ea〉	X	N	Z	V	C
	—	*	*	0	*

解説 ディスティネーション・オペランドを，1ビットだけ，左へローテート(回転)しますが，ローテート回数(1回)やオペレーションサイズ(ワード)が固定されています。

ローテートの様子を次に示します．

〈ea〉

C ← ディスティネーション オペランド

- オペランドを左へローテート．
- ローテートカウントは1に固定されている．
- 最上位ビットからローテートされ押し出されたビットは，Cビットだけでなく最下位ビットにも入る．
- Vビットは常にクリア(0)される．

CCR

X：変化せず

N：演算の結果，データの最上位ビット（MSB）が〝1〟ならセット(1)，それ以外はリセット（0）

Z：演算結果がゼロならセット（1)，それ以外はリセット（0）

V：常にリセット（0）

C：最後にローテートされて押し出されたビット値を保持

◉アドレッシング・モード

sou	size	(An)		(An)+		−(An)		d16(An)		d8(An,IX)		Abs.W		Abs.L	
		#	~	#	~	#	~	#	~	#	~	#	~	#	~
	B														
	W	2	12	2	12	2	14	4	16	4	18	4	16	6	20
	L														

(dest: (An) 〜 Abs.L)

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	1	dr	1	1	実効アドレス モード			レジスタ		

dr → 1：左ローテート

実効アドレス(メモリ・可変モード)

アドレッシングモード	5	4	3	2	1	0
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1

(対応ビット: 5〜0)

◉サンプル・リスト

```
ROL     (AØ)+
```

77 ●ROR [Rotate Right 右へローテート]

ROR {.B/.W/.L} Dn, Dn

X	N	Z	V	C
—	*	*	0	*

解説

ディスティネーション・オペランドを,ソース・オペランドのデータレジスタの内容だけ,右へローテート(回転)しますが,ローテートカウンタに使用されるソース・オペランドのデータレジスタ値は,64の余りが用いられ,0～63までの範囲となります.

ローテートの様子を次に示します.

Dn

ディスティネーション オペランド → C

- ●オペランドを右へローテート.
- ●最下位ビットからローテートされ押し出されたビットは,Cビットだけでなく最上位ビットにも入る.
- ●Vビットは常にクリア(0)される.

CCR

X：変化せず
N：演算の結果,データの最上位ビット(MSB)が"1"ならセット(1),それ以外はリセット(0)
Z：演算結果がゼロならセット(1),それ以外はリセット(0)
V：常にリセット(0)
C：最後にローテートされて押し出されたビット値を保持,ただし,ローテート・カウント値がゼロなら常にリセット(0)

●アドレッシング・モード

sou	size	dest Dn #	dest Dn ～
Dn	B	2	6+2n
	W	2	6+2n
	L	2	8+2n

●機械語フォーマット

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	1	1	0	カウントまたはレジスタ	dr	サイズ	i/r	1	1	レジスタ

- レジスタ → ディスティネーション・レジスタ番号(000～111)(操作されるデータレジスタの番号)
- i/r → 1：ローテート回数はソース・オペランドのデータレジスタで指定する
- サイズ → 00：バイト操作　01：ワード操作　10：ロングワード操作
- dr → 0：右ローテート
- カウントまたはレジスタ → ソースレジスタ番号(000～111)(ローテート回数が入っているデータレジスタ番号)

●サンプル・リスト

```
ROR      D2,D3
```

78 ●ROR [Rotate Right 右へローテート]

ROR {.B/.W/.L} #<data>, Dn	X	N	Z	V	C
	—	*	*	0	*

解説　ディスティネーション・オペランドを，ソース・オペランドのイミディエイト値だけ，右へローテート（回転）しますが，ローテートカウントとして指定できる範囲は1～8です．

ローテートの様子を次に示します．

Dn

ディスティネーションオペランド → C

- オペランドを右へローテート．
- ローテートカウントは1～8．
- 最下位ビットからローテートされ押し出されたビットは，Cビットだけでなく最上位ビットにも入る．
- Vビットは常にクリア（0）される．

CCR

X：変化せず

N：演算の結果，データの最上位ビット（MSB）が"1"ならセット（1），それ以外はリセット（0）

Z：演算結果がゼロならセット（1），それ以外はリセット（0）

V：常にリセット（0）

C：最後にローテートされて押し出されたビット値を保持

●アドレッシング・モード

sou	size	dest Dn #	dest Dn ～
#Imm	B	2	6+2n
	W	2	6+2n
	L	2	8+2n

●機械語フォーマット

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	1	1	0	カウントまたはレジスタ	dr	サイズ	i/r	1	1	レジスタ

- レジスタ → ディスティネーション・レジスタ番号(000～111)（操作されるデータレジスタの番号）
- i/r → 0：ローテート回数はソース・オペランドのイミディエイト値で指定する．
- サイズ → 00：バイト操作　01：ワード操作　10：ロングワード操作
- dr → 0：右ローテート
- カウントまたはレジスタ → シフト回数（1～8，ただし0は8を表す）

●サンプル・リスト

```
ROR        #2,D1
```

第3部 命令の詳細

79 ◉ROR [Rotate Right 右へローテート]

ROR {.W} <ea>	X	N	Z	V	C
	—	*	*	0	*

解説 ディスティネーション・オペランドを，1ビットだけ，右へローテート(回転)しますが，ローテート回数(1回)やオペレーションサイズ(ワード)が固定されています．

ローテートの様子を次に示します．

<ea>

ディスティネーション オペランド → C

- ●オペランドを右へローテート．
- ●ローテートカウントは1に固定されている．
- ●最下位ビットからローテートされ押し出されたビットは，Cビットだけでなく最上位ビットにも入る．
- ●Vビットは常にクリア(0)される．

CCR

X：変化せず

N：演算の結果，データの最上位ビット（MSB）が〝1〟ならセット(1)，それ以外はリセット（0）

Z：演算結果がゼロならセット（1），それ以外はリセット（0）

V：常にリセット（0）

C：最後にローテートされて押し出されたビット値を保持

◉アドレッシング・モード

sou	size	dest (An) #	(An) ~	(An)+ #	(An)+ ~	−(An) #	−(An) ~	d16(An) #	d16(An) ~	d8(An,IX) #	d8(An,IX) ~	Abs.W #	Abs.W ~	Abs.L #	Abs.L ~
	B														
	W	2	12	2	12	2	14	4	16	4	18	4	16	6	20
	L														

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	1	dr	1	1	実効アドレス モード			レジスタ		

dr → 0：右ローテート

実効アドレス(メモリ・可変モード)

アドレッシングモード	対応ビット 5	4	3	2	1	0
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1

◉サンプル・リスト

```
ROR      (A2)+
```

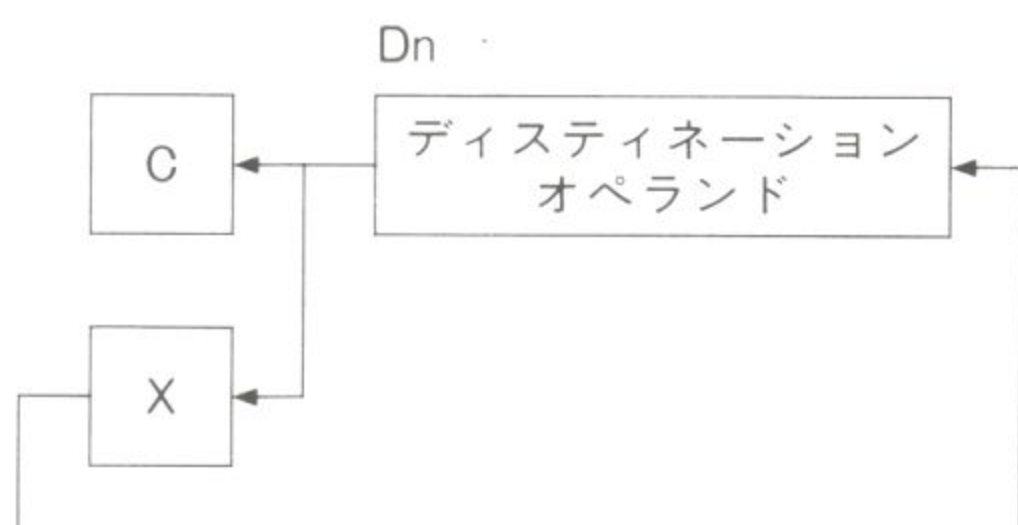
80●ROXL [Rotate with extend Left 拡張付き左ローテート]

ROXL {.B/.W/.L} Dn, Dn

X	N	Z	V	C
*	*	*	0	*

解説 ディスティネーション・オペランドを，ソース・オペランドのデータレジスタの内容だけ，Xビットも含めて，左へローテートしますが，ローテートカウンタに使用されるソースオペランドのデータレジスタ値は，64の余りが用いられ，0～63までの範囲となります．

ローテートの様子を次に示します．

Dn
C ← ディスティネーションオペランド ←
X

- ●オペランドをXビットも含めて左へローテート．
- ●最上位ビットからローテートされ押し出されたビットは，CビットとXビットへ入る．
- ●最下位ビットにはXビットの前の値が押し出されて入る．
- ●Vビットは常にクリア(0)される．

CCR

X：Cビットと同じ値であるが，ローテート・カウント値がゼロなら変化せず

N：演算の結果，データの最上位ビット（MSB）が"1"ならセット（1），それ以外はリセット（0）

Z：演算結果がゼロならセット(1)，それ以外はリセット（0）

V：常にリセット（0）

C：最後にローテートされて押し出されたビット値を保持。ただし，ローテート・カウント値がゼロなら，演算前のXビットの値を保持

●アドレッシング・モード

sou	size	dest Dn #	dest Dn ～
Dn	B	2	6+2n
	W	2	6+2n
	L	2	8+2n

●機械語フォーマット

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	1	1	0	カウントまたはレジスタ	dr	サイズ	i/r	1	0	レジスタ

- レジスタ → ディスティネーション・レジスタ番号(000～111)（操作されるデータレジスタの番号）
- i/r → 1：ローテート回数はソース・オペランドのデータレジスタで指定する．
- サイズ → 00：バイト操作　01：ワード操作　10：ロングワード操作
- dr → 1：左ローテート
- カウントまたはレジスタ → ソースレジスタ番号(000～111)（ローテート回数が入っているデータレジスタ番号）

●サンプル・リスト

```
ROXL    D2,D7
```

第3部 命令の詳細

81 ●ROXL [Rotate with extend Left 拡張付き左ローテート]

ROXL {.B/.W/.L} #<data>, Dn	X	N	Z	V	C
	*	*	*	0	*

解説

ディスティネーション・オペランドを，ソース・オペランドのイミディエイト値だけ，Xビットも含めて，左へローテートしますが，ローテートカウントとして指定できる範囲は1～8です．

ローテートの様子を次に示します．

Dn

C ← ディスティネーションオペランド ←

X

- ●オペランドをXビットも含めて左へローテート．
- ●ローテートカウントは1～8．
- ●最上位ビットからローテートされ押し出されたビットは，CビットとXビットへ入る．
- ●最下位ビットにはXビットの前の値が押し出されて入る．
- ●Vビットは常にクリア(0)される．

CCR

X：Cビットと同じ値

N：演算の結果，データの最上位ビット（MSB）が"1"ならセット（1），それ以外はリセット（0）

Z：演算結果がゼロならセット(1)，それ以外はリセット（0）

V：常にリセット（0）

C：最後にローテートされて押し出されたビット値を保持

●アドレッシング・モード

sou	size	dest Dn #	dest Dn ～
#Imm	B	2	6+2n
	W	2	6+2n
	L	2	8+2n

●機械語フォーマット

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	1	1	0	カウントまたはレジスタ	dr	サイズ	i/r	1	0	レジスタ

- レジスタ → ディスティネーション・レジスタ番号(000～111)（操作されるデータレジスタの番号）
- i/r → 0：ローテート回数はソース・オペランドのイミディエイト値で指定する．
- サイズ → 00：バイト操作 / 01：ワード操作 / 10：ロングワード操作
- dr → 1：左ローテート
- カウントまたはレジスタ → ローテート回数(1～8，ただし0は8を表す)

●サンプル・リスト

```
ROXL    #4,D7
```

82●ROXL [Rotate with extend Left　拡張付き左ローテート]

ROXL　{.W}　<ea>

X	N	Z	V	C
*	*	*	0	*

解説　ディスティネーション・オペランドを，1ビットだけ，Xビットも含めて，左へローテートしますが，ローテート回数(1回)やオペレーションサイズ(ワード)が固定されています．

ローテートの様子を次に示します．

<ea>

C ← ディスティネーション オペランド ←
X ←

- オペランドをXビットも含めて左へローテート．
- ローテートカウントは1に固定される．
- 最上位ビットからローテートされ押し出されたビットは，CビットとXビットへ入る．
- 最下位ビットにはXビットの前の値が押し出されて入る．
- Vビットは常にクリア(0)される．

CCR

X：Cビットと同じ値
N：演算の結果，データの最上位ビット（MSB）が"1"ならセット(1)，それ以外はリセット（0）
Z：演算結果がゼロならセット（1），それ以外はリセット（0）
V：常にリセット（0）
C：最後にローテートされて押し出されたビット値を保持

●アドレッシング・モード

sou	size	dest (An) #	(An) ~	(An)+ #	(An)+ ~	−(An) #	−(An) ~	d16(An) #	d16(An) ~	d8(An,IX) #	d8(An,IX) ~	Abs.W #	Abs.W ~	Abs.L #	Abs.L ~
	B														
	W	2	12	2	12	2	14	4	16	4	18	4	16	6	20
	L														

●機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	dr	1	1	実効アドレス モード			実効アドレス レジスタ		

dr → 1：左ローテート

実効アドレス(メモリ・可変モード)

アドレッシングモード	5	4	3	2	1	0
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1

●サンプル・リスト

```
ROXL    (AØ)+
```

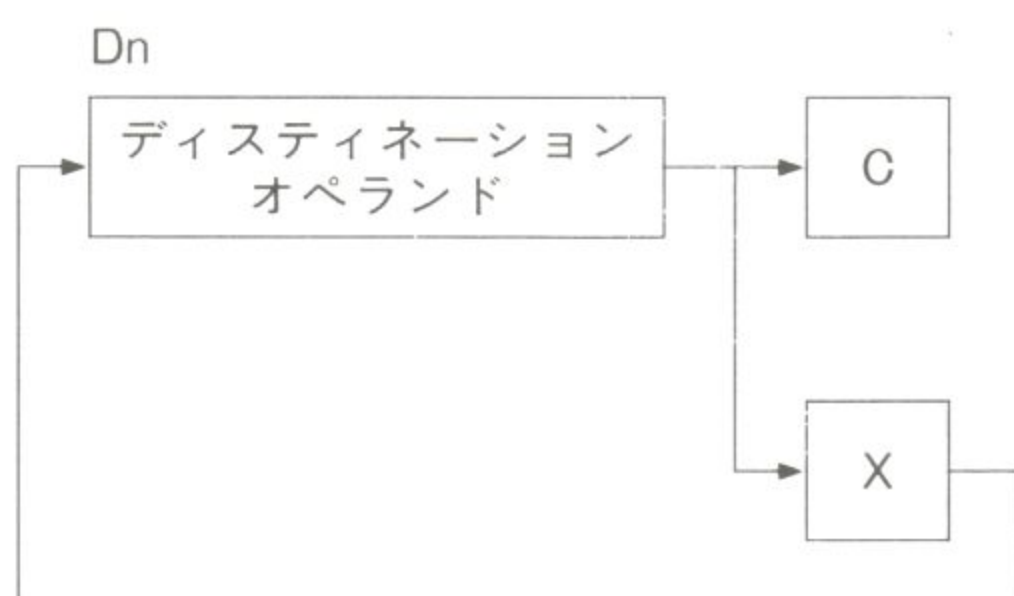
83 ◉ROXR [Rotate with extend Right 拡張付き右ローテート]

ROXR {.B/.W/.L} Dn, Dn	X	N	Z	V	C
	*	*	*	0	*

解説 ディスティネーション・オペランドを，ソース・オペランドのデータレジスタの内容だけ，Xビットも含めて，右へローテートしますが，ローテートカウンタに使用されるソースオペランドのデータレジスタ値は，64の余りが用いられ，0～63までの範囲となります．

ローテートの様子を次に示します．

Dn

ディスティネーションオペランド → C / X

- ●オペランドをXビットも含めて右へローテート．
- ●最下位ビットからローテートされ押し出されたビットは，CビットとXビットへ入る．
- ●最上位ビットにはXビットの前の値が押し出されて入る．
- ●Vビットは常にクリア(0)される．

CCR

X：Cビットと同じ値であるが，ローテート・カウント値がゼロなら変化せず

N：演算の結果，データの最上位ビット（MSB）が〝1〞ならセット（1），それ以外はリセット（0）

Z：演算結果がゼロならセット（1），それ以外はリセット（0）

V：常にリセット（0）

C：最後にローテートされて押し出されたビット値を保持，ただし，ローテート・カウント値がゼロなら，演算前のXビットの値を保持

◉アドレッシング・モード

sou	size	dest Dn #	～
Dn	B	2	6＋2n
	W	2	6＋2n
	L	2	8＋2n

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	カウントまたはレジスタ			dr	サイズ		i/r	1	0	レジスタ		

- レジスタ → ディスティネーション・レジスタ番号(000～111)(操作されるデータレジスタの番号)
- i/r → 1：ローテート回数はソース・オペランドのデータレジスタで指定する．
- サイズ → 00：バイト操作 01：ワード操作 10：ロングワード操作
- dr → 0：右ローテスト
- カウントまたはレジスタ → ソースレジスタ番号(000～111)（ローテート回数が入っているデータレジスタ番号）

◉サンプル・リスト

```
ROXR    D2,D7
```

84●ROXR [Rotate with extend Right 拡張付き右ローテート]

ROXR {.B/.W/.L} #<data>, Dn	X	N	Z	V	C
	*	*	*	0	*

解説　ディスティネーション・オペランドを，ソース・オペランドのイミディエイト値だけ，Xビットも含めて，右へローテートしますが，ローテートカウントとして指定できる範囲は1～8です．

ローテートの様子を次に示します．

Dn
ディスティネーション
オペランド
C
X

- オペランドをXビットも含めて右へローテート．
- ローテートカウントは1～8．
- 最下位ビットからローテートされ押し出されたビットは，CビットとXビットへ入る．
- 最上位ビットにはXビットの前の値が押し出されて入る．
- Vビットは常にクリア(0)される．

CCR

X：Cビットと同じ値
N：演算の結果，データの最上位ビット（MSB）が"1"ならセット（1），それ以外はリセット（0）
Z：演算結果がゼロならセット(1)，それ以外はリセット（0）
V：常にリセット（0）
C：最後にローテートされて押し出されたビット値を保持

●アドレッシング・モード

sou	size	dest Dn #	dest Dn ～
#Imm	B	2	6+2n
	W	2	6+2n
	L	2	8+2n

●機械語フォーマット

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	1	1	0	カウントまたはレジスタ	dr	サイズ	i/r	1	0	レジスタ

- レジスタ → ディスティネーション・レジスタ番号(000～111)(操作されるデータレジスタの番号)
- i/r → 0：ローテート回数はソースオペランドのイミディエイト値で指定する．
- サイズ → 00：バイト操作　01：ワード操作　10：ロングワード操作
- dr → 0：右ローテート
- カウントまたはレジスタ → ローテート回数（1～8，ただし0は8を表す）

●サンプル・リスト

```
ROXR    #4,D7
```

第3部 命令の詳細

85 ◉ROXR [Rotate with extend Right　拡張付き右ローテート]

ROXR　{.W}　<ea>	X	N	Z	V	C
	＊	＊	＊	0	＊

解説

ディスティネーション・オペランドを，1ビットだけ，Xビットも含めて，右へローテートしますが，ローテート回数(1回)やオペレーションサイズ(ワード)が固定されています．

ローテートの様子を次に示します．

<ea>

ディスティネーション オペランド → C

→ X

- オペランドをXビットも含めて右へローテート．
- ローテートカウントは1に固定される．
- 最下位ビットからローテートされ押し出されたビットは，CビットとXビットへ入る．
- 最上位ビットにはXビットの前の値が押し出されて入る．
- Vビットは常にクリア(0)される．

CCR

X：Cビットと同じ値

N：演算の結果，データの最上位ビット（MSB）が〝1″ならセット(1)，それ以外はリセット（0）

Z：演算結果がゼロならセット（1)，それ以外はリセット（0）

V：常にリセット（0）

C：最後にローテートされて押し出されたビット値を保持

◉アドレッシング・モード

sou	size	dest (An) #	(An) ~	(An)+ #	(An)+ ~	−(An) #	−(An) ~	d16(An) #	d16(An) ~	d8(An,IX) #	d8(An,IX) ~	Abs.W #	Abs.W ~	Abs.L #	Abs.L ~
	B														
	W	2	12	2	12	2	14	4	16	4	18	4	16	6	20
	L														

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	dr	1	1	実効アドレス モード			実効アドレス レジスタ		

dr → 0：右ローテート

実効アドレス(メモリ・可変モード)

アドレッシングモード	対応ビット 5	4	3	2	1	0
(An)	0	1	0	レジスタ番号		
(An)＋	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1

◉サンプル・リスト

```
ROXR    (AØ)+
```

86◉BTST [Test a Bit　ビット・テスト]

BTST　{.B/.L}　Dn,⟨ea⟩	X	N	Z	V	C
	—	—	*	—	—

解説

ディスティネーション・オペランドの指定ビットがゼロ(0)であるか否かをテストし，結果をZビットへ反映します．テストしたいビットは，ソース・オペランドのデータレジスタで指定します．

オペレーションサイズは，ディスティネーション・オペランドに依存し，ハードウエアが決定しますので，意味のあるビット番号を，ソースレジスタにセットしなければなりません．

▶ディスティネーション・オペランドをデータレジスタに指定した場合，オペレーションサイズはロングワードとなり，ビット番号には，ソース・オペランドの値の32の余り（0～31）が用いられます．

▶ディスティネーション・オペランドをメモリ上のデータに指定した場合，オペレーションサイズはバイトとなり，ビット番号には，ソース・オペランドの値の8の余り(0～7)が用いられます．

CCR

X：変化せず
N：変化せず
Z：テストしたビット番号の値がゼロならセット(1)，それ以外はリセット(0)
V：変化せず
C：変化せず

◉アドレッシング・モード

sou	size	dest																					
		Dn		(An)		(An)+		−(An)		d16(An)		d8(An,IX)		Abs.W		Abs.L		d16(PC)		d8(PC,IX)		CCR	
		#	~	#	~	#	~	#	~	#	~	#	~	#	~	#	~	#	~	#	~	#	~
Dn	B			2	8	2	8	2	10	4	12	4	14	4	12	6	16	4	12	4	14	4	10
	W																						
	L	2	6																				

注）MPUのデータシートにはCCR形式のマシンコードが存在するようだが，一般には使用する意味はない．

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	レジスタ			1	0	0	実効アドレス モード			実効アドレス レジスタ		

レジスタ → ソース・オペランドで指定したデータレジスタの番号で，テストするビット番号が格納されるデータレジスタの番号(000～111)

実効アドレス(データモード)

アドレッシングモード	対応ビット 5	4	3	2	1	0
Dn*	0	0	0	レジスタ番号		
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1
d16(PC)	1	1	1	0	1	0
d8(PC,IX)	1	1	1	0	1	1
CCR	1	1	1	1	0	0

＊サイズはロングワードのみをサポート．
その他のアドレッシンクはすべてバイトサイズであることに注意．

◉サンプル・リスト

```
BTST    DØ,D1
BTST    D1,(AØ)
```

第3部　命令の詳細

87 ◉BTST [Test a Bit　ビット・テスト]

BTST　{.B/.L}　#<data>,<ea>	X	N	Z	V	C
	—	—	*	—	—

解説

ディスティネーション・オペランドの指定ビットがゼロ(0)であるか否かをテストし，結果をZビットへ反映します．テストしたいビットは，ソース・オペランドのイミディエイト値で指定します．

オペレーションサイズは，ディスティネーション・オペランドに依存し，ハードウエアが決定しますので，意味のあるビット番号を，イミディエイト値で指定しなければなりません．

▶ディスティネーション・オペランドをデータレジスタに指定した場合，オペレーションサイズはロングワードとなり，ビット番号には，ソース・オペランドの値の32の余り（0～31）が用いられます．

▶ディスティネーション・オペランドをメモリ上のデータに指定した場合，オペレーションサイズはバイトとなり，ビット番号には，ソース・オペランドの値の8の余り(0～7)が用いられます．

CCR

X：変化せず
N：変化せず
Z：テストしたビット番号の値がゼロならセット（1），それ以外はリセット(0)
V：変化せず
C：変化せず

◉アドレッシング・モード

sou	size	dest																			
		Dn		(An)		(An)+		−(An)		d16(An)		d8(An,IX)		Abs.W		Abs.L		d16(PC)		d8(PC,IX)	
		#	～	#	～	#	～	#	～	#	～	#	～	#	～	#	～	#	～	#	～
#Imm	B			4	12	4	12	4	14	6	16	6	18	6	16	8	20	6	16	6	18
	W																				
	L	4	10																		

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	実効アドレス モード			実効アドレス レジスタ		

実効アドレス
(イミディエイト・データ形式を除くデータモード)

アドレッシングモード	対応ビット 5	4	3	2	1	0
Dn*	0	0	0	レジスタ番号		
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1
d16(PC)	1	1	1	0	1	0
d8(PC,IX)	1	1	1	0	1	1

〈第2ワード〉

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	#<data>							

ソース・オペランドで指定したビット番号

*サイズはロングワードのみをサポート．
その他のアドレッシングはすべてバイトサイズであることに注意．

◉サンプル・リスト

```
BTST    #4,D1
BTST    #2,(AØ)
```

88◉BSET [Test a Bit and Set ビット・テストとセット]

BSET {.B/.L} Dn,〈ea〉	X	N	Z	V	C
	—	—	*	—	—

解説

ディスティネーション・オペランドの指定ビットがゼロ(0)であるか否かをテストし，結果をZビットへ反映します．その後，指定ビットをセット(1)します．

セットしたいビットは，ソース・オペランドのデータレジスタで指定します．

オペレーションサイズは，ディスティネーション・オペランドに依存し，ハードウエアが決定しますので，意味のあるビット番号を，ソースレジスタにセットしなければなりません．

▶ディスティネーション・オペランドをデータレジスタに指定した場合，オペレーションサイズはロングワードとなり，ビット番号には，ソース・オペランドの値の32の余り（0〜31）が用いられます．

▶ディスティネーション・オペランドをメモリ上のデータに指定した場合，オペレーションサイズはバイトとなり，ビット番号には，ソース・オペランドの値の8の余り(0〜7)が用いられます．

CCR

X：変化せず
N：変化せず
Z：テストしたビット番号の値がゼロならセット（1），それ以外はリセット（0）
V：変化せず
C：変化せず

◉アドレッシング・モード

sou	size	dest Dn #	Dn ~	(An) #	(An) ~	(An)+ #	(An)+ ~	−(An) #	−(An) ~	d16(An) #	d16(An) ~	d8(An,IX) #	d8(An,IX) ~	Abs.W #	Abs.W ~	Abs.L #	Abs.L ~
Dn	B			2	12	2	12	2	14	4	16	4	18	4	16	6	20
	W																
	L	2	<8														

◉機械語フォーマット

15	14	13	12	11 10 9	8	7	6	5 4 3	2 1 0
0	0	0	0	レジスタ	1	1	1	実効アドレス モード	実効アドレス レジスタ

レジスタ → ソース・オペランドで指定したデータレジスタ番号で，テストするビット番号が格納されるデータレジスタの番号(000〜111)

実効アドレス(データ・可変モード)

アドレッシングモード	5	4	3	2 1 0
Dn*	0	0	0	レジスタ番号
(An)	0	1	0	レジスタ番号
(An)+	0	1	1	レジスタ番号
−(An)	1	0	0	レジスタ番号
d16(An)	1	0	1	レジスタ番号
d8(An,IX)	1	1	0	レジスタ番号
Abs.W	1	1	1	0 0 0
Abs.L	1	1	1	0 0 1

＊サイズはロングワードのみをサポート．
その他のアドレッシンクはすべてバイトサイズであることに注意．

◉サンプル・リスト

```
BSET    D1,D2
BSET    D7,(A2)
```

89 ◉BSET [Test a Bit and Set ビット・テストとセット]

BSET {.B/.L} ＃〈data〉,〈ea〉	X	N	Z	V	C
	—	—	*	—	—

解 説

ディスティネーション・オペランドの指定ビットがゼロ(0)であるか否かをテストし，結果をZビットへ反映します．その後，指定ビットをセット(1)します．

セットしたいビットは，ソース・オペランドのイミディエイト値で指定します．

オペレーションサイズは，ディスティネーション・オペランドに依存し，ハードウエアが決定しますので，意味のあるビット番号を，イミディエイト値で指定しなければなりません．

▶ディスティネーション・オペランドをデータレジスタに指定した場合，オペレーションサイズはロングワードとなり，ビット番号には，ソース・オペランドの値の32の余り（0～31）が用いられます．

▶ディスティネーション・オペランドをメモリ上のデータに指定した場合，オペレーションサイズはバイトとなり，ビット番号には，ソース・オペランドの値の8の余り（0～7）が用いられます．

CCR

X：変化せず
N：変化せず
Z：テストしたビット番号の値がゼロならセット（1），それ以外はリセット（0）
V：変化せず
C：変化せず

◉アドレッシング・モード

sou	size	dest Dn #	Dn ～	(An) #	(An) ～	(An)+ #	(An)+ ～	−(An) #	−(An) ～	d16(An) #	d16(An) ～	d8(An,IX) #	d8(An,IX) ～	Abs.W #	Abs.W ～	Abs.L #	Abs.L ～
＃Imm	B			4	16	4	16	4	18	6	20	6	22	6	20	8	24
	W																
	L	4	<12														

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	1	1	実効アドレス モード			実効アドレス レジスタ		

実効アドレス(データ・可変モード)

アドレッシングモード	対応ビット 5	4	3	2	1	0
Dn*	0	0	0	レジスタ番号		
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1

＊サイズはロングワードのみをサポート．その他のアドレッシングは，すべてバイトサイズであることに注意．

〈第2ワード〉

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	＃〈data〉							

ソース・オペランドで指定したビット番号

◉サンプル・リスト

```
BSET    #4,DØ
BSET    #2,(AØ)
```

90◉BCLR [Test a Bit and Clear　ビット・テストとクリア]

BCLR　{.B/.L}　Dn,〈ea〉	X	N	Z	V	C
	—	—	*	—	—

解説　ディスティネーション・オペランドの指定ビットがゼロ(０)であるか否かをテストし，結果をＺビットへ反映します．その後，指定ビットをクリア(０)します．

クリアしたいビットは，ソース・オペランドのデータレジスタで指定します．

オペレーションサイズは，ディスティネーション・オペランドに依存し，ハードウエアが決定しますので，意味のあるビット番号を，ソースレジスタにセットしなければなりません．

▶ディスティネーション・オペランドをデータレジスタに指定した場合，オペレーションサイズはロングワードとなり，ビット番号には，ソース・オペランドの値の32の余り（０～31）が用いられます．

▶ディスティネーション・オペランドをメモリ上のデータに指定した場合，オペレーションサイズはバイトとなり，ビット番号には，ソース・オペランドの値の８の余り(０～７)が用いられます．

CCR

X：変化せず
N：変化せず
Z：テストしたビット番号の値がゼロならセット（1），それ以外はリセット（0）
V：変化せず
C：変化せず

◉アドレッシング・モード

sou	size	dest Dn #	Dn ~	(An) #	(An) ~	(An)+ #	(An)+ ~	−(An) #	−(An) ~	d16(An) #	d16(An) ~	d8(An,IX) #	d8(An,IX) ~	Abs.W #	Abs.W ~	Abs.L #	Abs.L ~
Dn	B			2	12	2	12	2	14	4	16	4	18	4	17	6	20
	W																
	L	2	<10														

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	レジスタ			1	1	0	対応ビット モード			対応ビット レジスタ		

レジスタ → ソース・オペランドで指定したデータレジスタ番号で，テストするビット番号が格納されるデータレジスタの番号(000～111)

対応ビット → 実効アドレス(データ・可変モード)

アドレッシングモード	対応ビット 5	4	3	2	1	0
Dn*	0	0	0	レジスタ番号		
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1

＊サイズはロングワードのみをサポート．その他のアドレッシンクはすべてバイトサイズであることに注意．

◉サンプル・リスト

```
BCLR    DØ,D2
BCLR    D3,(AØ)
```

第3部 命令の詳細

91 ◉BCLR [Test a Bit and Clear　ビット・テストとクリア]

BCLR　{.B/.L}　#〈data〉,〈ea〉	X	N	Z	V	C
	—	—	*	—	—

解説

ディスティネーション・オペランドの指定ビットがゼロ（0）であるか否かをテストし，結果をZビットへ反映します．その後，指定ビットをクリア（0）します．

クリアしたいビットは，ソース・オペランドのイミディエイト値で指定します．

オペレーションサイズは，ディスティネーション・オペランドに依存し，ハードウエアが決定しますので，意味のあるビット番号を，イミディエイト値で指定しなければなりません．

▶ディスティネーション・オペランドをデータレジスタに指定した場合，オペレーションサイズはロングワードとなり，ビット番号には，ソース・オペランドの値の32の余り（0～31）が用いられます．

▶ディスティネーション・オペランドをメモリ上のデータに指定した場合，オペレーションサイズはバイトとなり，ビット番号には，ソース・オペランドの値の8の余り（0～7）が用いられます．

CCR

X：変化せず
N：変化せず
Z：テストしたビット番号の値がゼロならセット（1），それ以外はリセット（0）
V：変化せず
C：変化せず

◉アドレッシング・モード

sou	size	dest Dn #	Dn ～	(An) #	(An) ～	(An)+ #	(An)+ ～	−(An) #	−(An) ～	d16(An) #	d16(An) ～	d8(An,IX) #	d8(An,IX) ～	Abs.W #	Abs.W ～	Abs.L #	Abs.L ～
#Imm	B			4	16	4	16	4	18	6	20	6	22	6	20	8	24
	W																
	L	4	<14														

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	1	0	実効アドレス モード			実効アドレス レジスタ		

〈第2ワード〉

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	#〈data〉							

ソース・オペランドで指定したビット番号

実効アドレス（データ・可変モード）

アドレッシングモード	5	4	3	2	1	0
Dn*	0	0	0	レジスタ番号		
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1

＊サイズはロングワードのみをサポート．その他のアドレッシングはすべてバイトサイズであることに注意．

◉サンプル・リスト

```
BCLR    #2,DØ
BCLR    #1,(AØ)
```

92 ◉BCHG ［Test a Bit and Change　ビット・テストと反転］

BCHG　{.B/.L} Dn,<ea>	X	N	Z	V	C
	—	—	*	—	—

解説

ディスティネーション・オペランドの指定ビットがゼロ（0）であるか否かをテストし，結果をZビットへ反映します．その後，指定ビットを反転します．

反転したいビットは，ソース・オペランドのデータレジスタで指定します．

オペレーションサイズは，ディスティネーション・オペランドに依存し，ハードウエアが決定しますので，意味のあるビット番号を，ソースレジスタにセットしなければなりません．

▶ディスティネーション・オペランドをデータレジスタに指定した場合，オペレーションサイズはロングワードとなり，ビット番号には，ソース・オペランドの値の32の余り（0～31）が用いられます．

▶ディスティネーション・オペランドをメモリ上のデータに指定した場合，オペレーションサイズはバイトとなり，ビット番号には，ソース・オペランドの値の8の余り（0～7）が用いられます．

CCR

X：変化せず
N：変化せず
Z：テストしたビット番号の値がゼロならセット（1），それ以外はリセット（0）
V：変化せず
C：変化せず

◉アドレッシング・モード

sou	size	dest															
		Dn		(An)		(An)+		−(An)		d16(An)		d8(An,IX)		Abs.W		Abs.L	
		#	～	#	～	#	～	#	～	#	～	#	～	#	～	#	～
Dn	B			2	12	2	12	2	14	4	16	4	18	4	16	6	20
	W																
	L	2	<8														

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	レジスタ			1	0	1	実効アドレス					
										モード			レジスタ		

ソース・オペランドで指定したデータレジスタの番号で，テストするビット番号が格納されるデータレジスタの番号（000～111）

実効アドレス（データ・可変モード）

アドレッシングモード	対応ビット 5	4	3	2	1	0
Dn*	0	0	0	レジスタ番号		
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1

＊サイズはロングワードのみをサポート．
その他のアドレッシングはすべてバイトサイズであることに注意．

◉サンプル・リスト

```
BCHG    D2,D1
BCHG    D3,8(A0)
```

93 ◉BCHG [Test a Bit and Change　ビット・テストと反転]

BCHG {.B/.L} #<data>,<ea>	X	N	Z	V	C
	—	—	*	—	—

解説　ディスティネーション・オペランドの指定ビットがゼロ(0)であるか否かをテストし，結果をZビットへ反映します．その後，指定ビットを反転します．

反転したいビットは，ソース・オペランドのイミディエイト値で指定します．

オペレーションサイズは，ディスティネーション・オペランドに依存し，ハードウエアが決定しますので，意味のあるビット番号を，イミディエイト値で指定しなければなりません．

▶ディスティネーション・オペランドをデータレジスタに指定した場合，オペレーションサイズはロングワードとなり，ビット番号には，ソース・オペランドの値の32の余り（0～31）が用いられます．

▶ディスティネーション・オペランドをメモリ上のデータに指定した場合，オペレーションサイズはバイトとなり，ビット番号には，ソース・オペランドの値の8の余り(0～7)が用いられます．

CCR
X：変化せず
N：変化せず
Z：テストしたビット番号の値がゼロならセット（1），それ以外はリセット（0）
V：変化せず
C：変化せず

◉アドレッシング・モード

sou	size	dest															
		Dn		(An)		(An)+		−(An)		d16(An)		d8(An,IX)		Abs.W		Abs.L	
		#	~	#	~	#	~	#	~	#	~	#	~	#	~	#	~
#Imm	B			4	16	4	16	4	18	6	20	6	22	6	20	8	24
	W																
	L	4	<12														

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	1	実効アドレス					
										モード			レジスタ		

〈第2ワード〉

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	#<data>							

ソース・オペランドで指定したビット番号

実効アドレス(データ・可変モード)

アドレッシングモード	対応ビット					
	5	4	3	2	1	0
Dn*	0	0	0	レジスタ番号		
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1

*サイズはロングワードのみをサポート．その他のアドレッシングはすべてバイトサイズであることに注意．

◉サンプル・リスト

```
BCHG    #1,D2
BCHG    #4,(AØ)
```

94 ●ABCD [Add Decimal with extend　拡張付き10進加算]

ABCD　{.B}　Dn, Dn	X	N	Z	V	C
	*	U	*	U	*

解説

データレジスタ～データレジスタ間の2進化10進加算命令で，オペランドは指定したデータレジスタです．

ディスティネーション・オペランドに，ソース・オペランドとXビットを加算し，結果をディスティネーションのロケーションへ格納します．

CCR

X：Cビットと同じ値
N：未定義（結果は保証されない）
Z：演算結果がゼロでなければリセット（0），それ以外は変化せず
V：未定義（結果は保証されない）
C：桁上がり（10進キャリ）が発生すればセット（1），それ以外はリセット（0）

●アドレッシング・モード

sou	size	dest Dn #	dest Dn ～
Dn	B	2	6
	W		
	L		

●機械語フォーマット

15	14	13	12	11 10 9	8	7	6	5	4	3	2 1 0
1	1	0	0	ディスティネーションレジスタ	1	0	0	0	0	R/M	ソースレジスタ

- ソースレジスタ → ソース側のデータレジスタ番号（000～111）
- R/M → 0：データレジスタ間の演算
- ディスティネーションレジスタ → ディスティネーション側のデータレジスタ番号（000～111）

●サンプル・リスト

```
ABCD    D2,D4
```

95 ●ABCD [Add Decimal with extend 拡張付き10進加算]

ABCD {.B} −(An),−(An)	X	N	Z	V	C
	*	U	*	U	*

解説

メモリ～メモリ間の2進化10進加算命令で，オペランドは，プリデクリメント・アドレスレジスタ間接形式でポイントされるメモリ上に存在します．

ディスティネーション・オペランドに，ソース・オペランドとXビットを加算し，結果をディスティネーションのロケーションへ格納します．

CCR

X：Cビットと同じ値
N：未定義（結果は保証されない）
Z：演算結果がゼロでなければリセット(0)，それ以外は変化せず
V：未定義（結果は保証されない）
C：桁上がり（10進キャリ）が発生すればセット（1），それ以外はリセット（0）

●アドレッシング・モード

sou	size	dest −(An) #	dest −(An) ～
−(An)	B	2	18
	W		
	L		

●機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	ディスティネーション レジスタ			1	0	0	0	0	R/M	ソース レジスタ		

- ソースレジスタ → プリデクリメント・アドレス形式で使用する，ソース側のアドレスレジスタ番号(000～111)
- R/M → 1：メモリ間の演算
- ディスティネーションレジスタ → プリデクリメント・アドレス形式で使用する，ディスティネーション側のアドレスレジスタ番号(000～111)

●サンプル・リスト

```
ABCD    -(A2),-(A6)
```

96 ◉SBCD [Subtract Decimal with extend 拡張付き10進減算]

SBCD {.B} Dn, Dn	X	N	Z	V	C
	*	U	*	U	*

解説

データレジスタ～データレジスタ間の2進化10進減算命令で，オペランドは指定したデータレジスタです．

ディスティネーション・オペランドから，ソース・オペランドとXビットを減算し，結果をディスティネーションのロケーションへ格納します．

CCR

X：Cビットと同じ値
N：未定義（結果は保証されない）
Z：演算結果がゼロでなければリセット（0），それ以外は変化せず
V：未定義（結果は保証されない）
C：桁下がり（10進ボロー）が発生すればセット（1），それ以外はリセット（0）

◉アドレッシング・モード

sou	size	dest Dn #	dest Dn ～
Dn	B	2	6
	W		
	L		

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	ディスティネーション レジスタ			1	0	0	0	0	R/M	ソース レジスタ		

- ソース側のデータレジスタ番号(000～111)
- 0：データレジスタ間の演算
- ディスティネーション側のデータレジスタ番号(000～111)

◉サンプル・リスト

```
SBCD    D2,D3
```

第3部 命令の詳細

97 ●SBCD [Subtract Decimal with extend 拡張付き10進減算]

SBCD {.B} −(An), −(An)	X	N	Z	V	C
	*	U	*	U	*

解説

メモリ～メモリ間の2進化10進減算命令で，オペランドは，プリデクリメント・アドレスレジスタ間接形式でポイントされるメモリ上に存在します．

ディスティネーション・オペランドから，ソース・オペランドとXビットを減算し，結果をディスティネーションのロケーションへ格納します．

CCR

X：Cビットと同じ値
N：未定義（結果は保証されない）
Z：演算結果がゼロでなければリセット（0），それ以外は変化せず
V：未定義（結果は保証されない）
C：桁下がり（10進ボロー）が発生すればセット（1），それ以外はリセット（0）

●アドレッシング・モード

sou	size	dest −(An) #	dest −(An) ～
−(An)	B	2	18
	W		
	L		

●機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	ディスティネーション レジスタ			1	0	0	0	0	R/M	ソース レジスタ		

- ソースレジスタ → プリデクリメント・アドレス形式で使用する，ソース側のアドレスレジスタ番号(000～111)
- R/M → 1：メモリ間の演算
- ディスティネーションレジスタ → プリデクリメント・アドレス形式で使用する，ディスティネーション側のアドレスレジスタ番号(000～111)

●サンプル・リスト

```
SBCD    -(A2),-(A6)
```

98 ◉NBCD [Negate Decimal with extend 拡張付き10進符号反転]

NBCD {.B} 〈ea〉	X	N	Z	V	C
	*	U	*	U	*

解説 オペランドの補数をとる命令で，演算は2進化10演算で行われます．
ゼロ(0)から，ディスティネーション・オペランドとXビットを減算し，結果をディスティネーションのロケーションへ格納しますが，Xビットがリセット（0）されていれば10の補数，セット（1）されていれば9の補数が求まります．

CCR
X：Cビットと同じ値
N：未定義（結果は保証されない）
Z：演算結果がゼロでなければリセット(0)，それ以外は変化せず
V：未定義（結果は保証されない）
C：桁下がり（10進ボロー）が発生すればセット（1），それ以外はリセット（0）

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	0	実効アドレス					
										モード			レジスタ		

実効アドレス(データ・可変モード)

アドレッシングモード	対応ビット 5	4	3	2	1	0
Dn	0	0	0	レジスタ番号		
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An, IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1

◉アドレッシング・モード

sou	size	Dn #	Dn ~	(An) #	(An) ~	(An)+ #	(An)+ ~	−(An) #	−(An) ~	d16(An) #	d16(An) ~	d8(AnIX) #	d8(AnIX) ~	Abs.W #	Abs.W ~	Abs.L #	Abs.L ~
	B	2	6	2	12	2	12	2	14	4	16	4	18	4	16	6	20
	W																
	L																

＊ソース・オペランドは存在しない

◉サンプル・リスト

```
NBCD    D2
NBCD    (A0)
```

第3部 命令の詳細

99 ●Bcc [Branch Conditionally 条件付き分岐]

Bcc {.B/.W} <label>	X	N	Z	V	C
	—	—	—	—	—

解説

条件付き分岐命令です．

命令のcc部で指定された条件が満たされた場合，指定したラベルへ分岐し，それ以外は分岐せず，本命令の次に位置する命令へ制御が移行します．

本命令の直前で比較命令を実行し，それが等しい場合にLABELという記号で表現されるロケーションへ分岐させるには，

BEQ LABEL

と記述します．このように，条件分岐命令は，CCRに影響をおよぼす命令の後を受け，必要なロケーションへ制御を移行させるために使用します．

条件が満足された場合，(PC)＋ディスプレースメント，に分岐しますが，分岐先までの相対距離がどのように生成されるか，に関しては，アセンブラが計算しますから，プログラマが関知する必要はありません。ただし，本命令と分岐先までの相対距離は2バイトのオフセットで表現され，それ以上離れたロケーションである場合，アセンブラからエラーメッセージが出力されます．

ccとして以下の条件を指定することができますが，プログラミング・レベルで頻繁に使用される大小表現と，その他の表現に区分けしています．

CMP x,y (y－xを実行している)

条件式	表現		その他の表現
	符号なし	符号付き	
y ＞ x	HI(High)	GT(Greater Than)	PL(Plus)
y ≧ x	CC(Carry Clear)	GE(Greater or Equal)	MI(Minus)
y ＝ x	EQ(Equal)	EQ(Equal)	VC(Overflow Clear)
y ≠ x	NE(Not Equal)	NE(Not Equal)	VS(Overflow Set)
y ＜ x	CS(Carry Set)	LT(Less Than)	
y ≦ x	LS(Lower or Same)	LE(Less or Equal)	

CCR

X：変化せず
N：変化せず
Z：変化せず
V：変化せず
C：変化せず

●アドレッシング・モード

sou	size	dest disp #	dest disp ～
	B	2	10
	W	2	8
	L		

(注)アドレッシング形式はプログラムカウンタ・リラティブに分類されるが，書式の関係からdispという分類をした．

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	コンディション				8ビット・ディスプレースメント							

8ビット・ディスプレースメント → 2の補数の整数で，この命令と分岐先の命令との相対距離を表し，バイト単位である．

コンディション →

ニーモニック	対応ビット			
	11	10	9	8
CC	0	1	0	0
CS	0	1	0	1
EQ	0	1	1	1
GE	1	1	0	0
GT	1	1	1	0
HI	0	0	1	0
LE	1	1	1	1
LS	0	0	1	1
LT	1	1	0	1
MI	1	0	1	1
NE	0	1	1	0
PL	1	0	1	0
VC	1	0	0	0
VS	1	0	0	1

〈第2ワード〉

第2ワードが作成される時，第1ワードのディスプレースメント・フィールドには，ゼロが代入されます．

15 ～ 0
16ビット・ディスプレースメント

16ビット・ディスプレースメント → 2の補数の整数で，この命令と分岐先の命令との相対距離を表す．

(注1) 第1ワードの8ビット・ディスプレースメントが有効であれば，すなわち，ゼロでない場合，第2ワードは作成されません．これは8ビットのディスプレースメントで分岐可能であるため，作成する必要がないことを意味します．

(注2) PCの値は，この命令の位置+2に更新されているので，ハンドアセンブルで相対距離の計算をする時には，注意すること．

(注3) 本命令の直後に分岐するような分岐そのものは無意味であるわけですが，このような命令フォーマットは存在しません．8ビットディスプレースメントがゼロであれば，それはワード分岐となるからです．

◉サンプル・リスト

```
BHI     EXE_JOB
BPL     EXE_JOB
BEQ     EXE_JOB
```

EXE_JOBは記号番地を意味する

100 ●DBcc [Test Condition, Decrement, and Branch 条件テスト，デクリメントおよび分岐]

DBcc {.W} Dn,<label>	X	N	Z	V	C
	—	—	—	—	—

解説

ループの制御命令です．

命令のcc部，データレジスタ，ラベル（ディスプレースメント），の3つのパラメータによって，ある命令群を繰り返し実行するもので，68000は本命令に出会うと，次のように行動します（分岐先のシンボルをLABELとする）．

① cc部で指定されているループの終了条件をテストする；
- ●条件を満足していれば，ループを抜け出す（本命令の次に位置する命令を実行）．
- ●それ以外は ②へ．

② ループカウンタであるDnの内容を1つだけ減じる（デクリメント）；
- ●Dnが−1（マイナス1）ならループを抜け出す（本命令の次に位置する命令を実行）．
- ●それ以外はLABELという記号で表現されるロケーションへ分岐する．

分岐条件が満足された場合，(PC)＋ディスプレースメント，に分岐しますが，分岐先までの相対距離がどのように生成されるか，に関しては，アセンブラが計算しますから，プログラマが関知する必要はありません．ただし，本命令と分岐先までの相対距離は2バイトのオフセットで表現され，それ以上離れたロケーションである場合，アセンブラからエラーメッセージが出力されます．

本命令は，「Dnで指定した文字列長の読み込みを実行するが，c/r（キャリッジ・リターン・コード）を見つけたら処理を終了する」というように，終了条件が複数の場合に威力を発揮します．いうまでもなく，このときのLABELで表現されるロケーションには，1文字読み込み処理のエントリが位置しているはずです．

ccとして以下の条件を指定することができますが，プログラミング・レベルで頻繁に使用される大小表現と，その他の表現に区分けしています．

CMP x,y (y−xを実行している)

条件式	表現		その他の表現
	符号なし	符号付き	
y > x	HI(High)	GT(Greater Than)	PL(Plus)
y ≧ x	CC(Carry Clear)	GE(Greater or Equal)	MI(Minus)
y = x	EQ(Equal)	EQ(Equal)	VC(Overflow Clear)
y ≠ x	NE(Not Equal)	NE(Not Equal)	VS(Overflow Set)
y < x	CS(carry Set)	LT(Less Than)	F*(always False)
y ≦ x	LS(Lower or Same)	LE(Less or Equal)	T(always True)

*DBFという表現はスマートではないので，DBRAというニーモニックが許され，これは，単にDnにカウント値をセットしてループ制御するような，単純ループに使用される（このような用途も頻繁に存在する）．

CCR

X：変化せず
N：変化せず
Z：変化せず
V：変化せず
C：変化せず

◉アドレッシング・モード

sou	size	dest disp #	dest disp ~
Dn	B		
	W	4	•
	L		

（注） アドレッシング形式はプログラムカウンタ・リラティブに分類されるが，書式の関係からdispという分類をした．

~	cc	Counter	Branch
10	false	±1	yes
12	true	±1	no
14	false	expired	no

◉機械語フォーマット

15	14	13	12	11 10 9 8	7	6	5	4	3	2 1 0
0	1	0	1	コンディション	1	1	0	0	1	レジスタ

レジスタ → カウンタとなるデータレジスタ番号(000～111)

ニーモニック	対応ビット 11	10	9	8
CC	0	1	0	0
CS	0	1	0	1
EQ	0	1	1	1
F	0	0	0	1
GE	1	1	0	0
GT	1	1	1	0
HI	0	0	1	0
LE	1	1	1	1
LS	0	0	1	1
LT	1	1	0	1
MI	1	0	1	1
NE	0	1	1	0
PL	1	0	1	0
T	0	0	0	0
VC	1	0	0	0
VS	1	0	0	1

<第2ワード>

15 ～ 0
16ビット・ディスプレースメント

2の補数の整数で，この命令と分岐先との相対距離を表す．

（注） PCの値は，この命令の位置+2に更新されているので，ハンドアセンブルで相対距離の計算をする時には，注意すること．

◉サンプル・リスト

```
DBCC    D2,EXE_JOB
DBCS    D4,EXE_JOB
DBVC    D5,EXE_JOB
DBRA    D1,EXE_JOB
```

EXE_JOBは記号番地を意味する．

101 ●Scc [Set According to Condition　条件付きセット]

Scc {.B} <ea>	X	N	Z	V	C
	—	—	—	—	—

解説

条件セット命令です．

本命令は，cc部で指定した条件が満足されると，オペランドの全ビットをセット（$FF）にし，そうでなければ，リセット（$00）するというものです．なお，オペランドのサイズはバイトです．

たとえば，ある値の比較結果が等しいかどうかを，D0に記憶しておきたければ，

SEQ　D0

のような記述をします．

ccとして以下の条件を指定することができますが，プログラミング・レベルで頻繁に使用される大小表現と，その他の表現に区分けしています．

CMP　x, y　（y−xを実行している）

条件式	表現		その他の表現
	符号なし	符号付き	
y > x	HI(High)	GT(Greater Than)	PL(Plus)
y ≧ x	CC(Carry Clear)	GE(Greater or Equal)	MI(Minus)
y = x	EQ(Equal)	EQ(Equal)	VC(Overflow Clear)
y ≠ x	NE(Not Equal)	NE(Not Equal)	VS(Overflow Set)
y < x	CS(Carry Set)	LT(Less Than)	F(always False)
y ≦ x	LS(Lower or Same)	LE(Less or Equal)	T(always True)

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	コンディション				1	1	実効アドレス					
										モード			レジスタ		

ニーモニック	対応ビット			
	11	10	9	8
CC	0	1	0	0
CS	0	1	0	1
EQ	0	1	1	1
F	0	0	0	1
GE	1	1	0	0
GT	1	1	1	0
HI	0	0	1	0
LE	1	1	1	1
LS	0	0	1	1
LT	1	1	0	1
MI	1	0	1	1
NE	0	1	1	0
PL	1	0	1	0
T	0	0	0	0
VC	1	0	0	0
VS	1	0	0	1

実効アドレス（データ・可変モード）

アドレッシングモード	対応ビット					
	5	4	3	2	1	0
Dn	0	0	0	レジスタ番号		
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1

CCR

X：変化せず
N：変化せず
Z：変化せず
V：変化せず
C：変化せず

◉アドレッシング・モード

sou	size	dest															
		Dn		(An)		(An)+		−(An)		d16(An)		d8(An,IX)		Abs.W		Abs.L	
		#	~	#	~	#	~	#	~	#	~	#	~	#	~	#	~
	B	2	●	2	12	2	12	2	14	4	16	4	18	4	16	6	20
	W																
	L																

~	条件
4	false
6	true

＊ソース・オペランドは存在しない

◉サンプル・リスト

```
SGT     D2
SEQ     (A1)
SLS     FLG_AREA
```

FLG＿AREAは記号番地を意味する

第3部 命令の詳細

102 ◉BRA [Branch Always 無条件分岐]

BRA {.B/.W} <label>	X	N	Z	V	C
	—	—	—	—	—

解説 無条件ブランチ命令であり，ラベルで指定したロケーションへ無条件に分岐します．
(PC)＋ディスプレースメント，に分岐しますが，分岐先までの相対距離がどのように生成されるか，に関しては，アセンブラが計算しますから，プログラマが関知する必要はありません．ただし，本命令と分岐先までの相対距離は2バイトのオフセットで表現され，それ以上離れたロケーションである場合，アセンブラからエラーメッセージが出力されます．

CCR
X：変化せず
N：変化せず
Z：変化せず
V：変化せず
C：変化せず

◉アドレッシング・モード

sou	size	dest disp #	dest disp ～
	B	2	10
	W	4	10
	L		

(注) アドレッシング形式はプログラムカウンタ・リラティブに分類されるが，書式の関係からdispという分類をした．

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	0	0	8ビット・ディスプレースメント							

→ 2の補数の整数で，この命令と分岐先の命令との相対距離を表し，バイト単位である．

<第2ワード>

第2ワードが作成される時，第1ワードのディスプレースメント・フィールドには，ゼロが代入されます．

15 ～ 0
16ビット・ディスプレースメント

→ 2の補数の整数で，この命令と分岐先の命令との相対距離を表す．

(注1) 第1ワードの8ビット・ディスプレースメントが有効であれば，すなわち，ゼロでない場合，第2ワードは作成されません．
これは8ビットのディスプレースメントで分岐可能であるため，作成する必要がないことを意味します．
(注2) PCの値は，この命令の位置＋2に変更されているので，ハンドアセンブルで相対距離の計算をする時には，注意すること．
(注3) 本命令の直後に分岐するような分岐そのものは無意味であるわけですが，このような命令フォーマットは存在しません．8ビットディスプレースメントがゼロであれば，それはワード分岐となるからです．

◉サンプル・リスト

```
BRA      EXE_JOB  ←── EXE_JOBは記号番地を意味する
```

103 ◉BSR [Branch to Subroutine　サブルーチン分岐]

BSR　{.B/.W}　<label>	X	N	Z	V	C
	―	―	―	―	―

解説

サブルーチン分岐命令です．BSR命令直後の命令のアドレスをシステム・スタックへプッシュし，ラベルで指定したロケーションへ分岐します．

(PC)＋ディスプレースメント，に分岐しますが，分岐先までの相対距離がどのように生成されるか，に関しては，アセンブラが計算しますから，プログラマが関知する必要はありません．ただし，本命令と分岐先までの相対距離は2バイトのオフセットで表現され，それ以上離れたロケーションである場合，アセンブラからエラーメッセージが出力されます．

本命令は，もどり番地をシステムスタックにプッシュしており，RTSやRTRなどのリターン命令と組み合わせ，サブルーチン・コールを行うためのものです．

CCR

X：変化せず
N：変化せず
Z：変化せず
V：変化せず
C：変化せず

◉アドレッシング・モード

sou	size	dest disp #	dest disp ～
	B	2	20
	W	4	20
	L		

(注)　アドレッシング形式はプログラムカウンタ・リラティブに分類されるが，書式の関係から dispという分類をした．

◉機械語フォーマット

15	14	13	12	11	10	9	8	7–0
0	1	1	0	0	0	0	1	8ビット・ディスプレースメント

→ 2の補数の整数で，この命令と分岐先の命令との相対距離を表し，バイト単位である．

〈第2ワード〉

第2ワードが作成される時，第1ワードのディスプレースメント・フィールドには，ゼロが代入されます．

15–0
16ビット・ディスプレースメント

→ 2の補数の整数で，この命令と分岐先の命令との相対距離を表す．

(注1)　第1ワードの8ビット・ディスプレースメントが有効であれば，すなわち，ゼロでない場合，第2ワードは作成されません．
これは8ビットのディスプレースメントで分岐可能であるため，作成する必要がないことを意味します．
(注2)　PCの値は，この命令の位置＋2に更新されているので，ハンドアセンブルで相対距離の計算をする時には，注意すること．
(注3)　本命令の直後に分岐するような分岐そのものは無意味であるわけですが，このような命令フォーマットは存在しません．8ビットディスプレースメントがゼロであれば，それはワード分岐となるからです．

◉サンプル・リスト

```
	BSR	EXE_JOB	←─EXE_JOBは記号番地を意味する
```

104◉JMP [Jump ジャンプ]

JMP 〈ea〉	X	N	Z	V	C
	—	—	—	—	—

解説 無条件分岐命令ですが，ブランチ命令と異なり，68000のサポートする全域へ制御を移行できます．

今，A0に＄2000というアドレスデータが格納されているとき，

JMP (A0)

によって，プログラムの制御は無条件にアドレス＄2000へ移行します．

CCR

X：変化せず
N：変化せず
Z：変化せず
V：変化せず
C：変化せず

◉アドレッシング・モード

sou	size	dest (An) #	(An) ~	d16(An) #	d16(An) ~	d8(An,IX) #	d8(An,IX) ~	Abs.W #	Abs.W ~	Abs.L #	Abs.L ~	d16(PC) #	d16(PC) ~	d8(PC,IX) #	d8(PC,IX) ~
		2	8	4	10	4	14	4	10	6	12	4	10	4	14

＊サイズは存在しない．

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	1	1	実効アドレス モード			実効アドレス レジスタ		

実効アドレス（制御モード）

アドレッシングモード	対応ビット 5	4	3	2	1	0
(An)	0	1	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1
d16(PC)	1	1	1	0	1	0
d8(PC,IX)	1	1	1	0	1	1

◉サンプル・リスト

```
JMP     (AØ)
JMP     EXE_JOB ←─EXE_JOBは記号番地を意味する
```

105 ◉JSR [Jump to Subroutine　サブルーチン・ジャンプ]

JSR 〈ea〉

X	N	Z	V	C
—	—	—	—	—

解説

サブルーチン・コール命令ですが，ブランチ命令と異なり，68000のサポートする全域へ制御を移行できます．

今，A 0 に＄2000というアドレスデータが格納されているとき，

JSR　(A0)

によって，68000はJSR直後の命令が置かれているアドレスをシステム・スタックにプッシュし，アドレス＄2000へ分岐します．

本命令は,もどり番地をシステム・スタックにプッシュしており,RTSやRTRなどのリターン命令と組み合わせ，サブルーチン・コールを行うためのものです．

CCR

X：変化せず
N：変化せず
Z：変化せず
V：変化せず
C：変化せず

◉アドレッシング・モード

sou	size	dest (An) #	(An) ~	d16(An) #	d16(An) ~	d8(An,IX) #	d8(An,IX) ~	Ads.W #	Ads.W ~	Abs.L #	Abs.L ~	d16(PC) #	d16(PC) ~	d8(PC,IX) #	d8(PC,IX) ~
		2	16	4	18	4	22	4	18	6	20	4	18	4	22

*サイズは存在しない

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	1	0	実効アドレス モード			実効アドレス レジスタ		

実効アドレス(制御モード)

アドレッシングモード	対応ビット 5	4	3	2	1	0
(An)	0	1	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1
d16(PC)	1	1	1	0	1	0
d8(PC,IX)	1	1	1	0	1	1

◉サンプル・リスト

```
JSR     (AØ)
JSR     EXE_JOB ←─EXE_JOBは記号番地を意味する
```

106●RTR [Return and Restore Condition Codes　CCR回復とリターン]

RTR	X	N	Z	V	C
	*	*	*	*	*

解説　サブルーチンからの復帰命令で，CCRと戻り番地を意味するプログラムカウンタ(PC)を，システムスタックから取り出します．

命令実行前のCCRとPCの内容は失われますが，ステータスレジスタのスーパバイザ部(上位8ビット)は影響を受けません．

note：PCの内容が書き換わるということは，PCでポイントされるアドレスへ制御が移行するのだから，分岐が起こることを意味する．

CCR

X：スタック上のワードに対応するビット4の値
N：スタック上のワードに対応するビット3の値
Z：スタック上のワードに対応するビット2の値
V：スタック上のワードに対応するビット1の値
C：スタック上のワードに対応するビット0の値

◉アドレッシング・モード

sou	size	dest	
		#	~
		2	20

*ソース・オペランド，ディスティネーション・オペランドという概念は存在しない．

*サイズは存在しない．

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	1	1	1

◉サンプル・リスト

```
RTR
```

107 ◉RTS [Return from Subroutine サブルーチンからのリターン]

RTS	X	N	Z	V	C
	—	—	—	—	—

解説 サブルーチンからの復帰命令で，戻り番地を意味するプログラムカウンタ（PC）を，システムスタックから取り出します．命令実行前のPCの内容は失われます．

note：PCの内容が書き換わるということは，PCでポイントされるアドレスへ制御が移行するのだから，分岐が起こることを意味する．

CCR
X：変化せず
N：変化せず
Z：変化せず
V：変化せず
C：変化せず

◉アドレッシング・モード

sou	size	dest	
		＃	～
		2	16

＊ソース・オペランド，ディスティネーション・オペランドという概念は存在しない．

＊サイズは存在しない．

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	1	0	1

◉サンプル・リスト

```
RTS
```

108 ◉TRAP [Trap トラップ]

TRAP #<ベクタ番号>	X	N	Z	V	C
	—	—	—	—	—

解説

プロセッサは例外処理を開始します．

本命令については，プロセッサの例外処理に関する的確な知識が要求されます．

指定ベクタ番号（0～15を指定する）と例外処理のエントリアドレスを保持する先頭アドレス（例外ベクタと呼ぶ）との対応表は，次に示す通りです．

#<ベクタ>	アドレス(HEX)
0	$80
1	$84
2	$88
3	$8C
4	$90
5	$94
6	$98
7	$9C
8	$A0
9	$A4
10	$A8
11	$AC
12	$B0
13	$B4
14	$B8
15	$BC

CCR

X：変化せず
N：変化せず
Z：変化せず
V：変化せず
C：変化せず

◉アドレッシング・モード

sou	size	dest #	dest ～
#<ベクタ番号>		2	34

*サイズ，ソース／ディスティネーションという概念はまったく存在しない．

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	0	0	ベクタ番号			

→ ベクタ番号としては，0～15までの16通りを指定できる．

◉サンプル・リスト

```
TRAP    #15
```

109 ◉TRAPV [Trap on Overflow　オーバフロー・トラップ]

TRAPV

X	N	Z	V	C
—	—	—	—	—

解説　Vビット（オーバフロー・ビット）がセットされている状態で本命令を実行すると，プロセッサは例外処理を開始しますが，そうでない場合は何もせず，単に次の命令に制御が移ります．

本命令による例外処理のエントリアドレスを保持する先頭アドレス（例外ベクタと呼ぶ）は，$1Cです．

CCR

X：変化せず
N：変化せず
Z：変化せず
V：変化せず
C：変化せず

◉アドレッシング・モード

sou	size	dest	
		#	~
		2	34
		2	4

＊サイズ，ソース／ディスティネーションという概念は，まったく存在しない．

2 34 ←トラップが発生した時
2 4 ←発生しなかった時

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	1	1	0

◉サンプル・リスト

```
TRAPV
```

110 ◉CHK [Check Register Against Bounds レジスタ境界チェック]

CHK {.W} 〈ea〉, Dn

X	N	Z	V	C
—	*	U	U	U

解説　ディスティネーション・オペランドとして指定したデータレジスタの下位ワードとゼロ(０)を比較し，さらにソース・オペランドの上限値(下位ワード)を比較します．ただし，上限値は２の補数表現です．

比較結果によって例外処理を開始しますが，該当ケースは次の２つに限定され，３番目のケースではトラップは発生せず，単に次の命令に制御が移ります．

① Dn＜0　　……Nビットをセットして例外処理を開始
② Dn＞(〈ea〉)……Nビットをクリアして例外処理を開始
③ 0≦Dn≦(〈ea〉)……例外処理は開始されない

本命令による例外処理のエントリアドレスを保持する先頭アドレス(例外ベクタと呼ぶ)は，$18です．

CCR
X：変化せず
N：Dn＜0ならセット（1）、Dn＞（＜ea＞）ならリセット（0），それ以外は未定義(結果は保証されない)
Z：未定義（結果は保証されない）
V：未定義（結果は保証されない）
C：未定義（結果は保証されない）

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	レジスタ			1	1	0	実効アドレス					
										モード			レジスタ		

実効アドレス(データモード)

アドレッシングモード	対応ビット 5	4	3	2	1	0
Dn	0	0	0	レジスタ番号		
(An)	0	1	0	レジスタ番号		
(An)+	0	1	1	レジスタ番号		
−(An)	1	0	0	レジスタ番号		
d16(An)	1	0	1	レジスタ番号		
d8(An,IX)	1	1	0	レジスタ番号		
Abs.W	1	1	1	0	0	0
Abs.L	1	1	1	0	0	1
d16(PC)	1	1	1	0	1	0
d8(PC,IX)	1	1	1	0	1	1
#Imm	1	1	1	1	0	0

レジスタ：ディスティネーション・オペランドとして指定したデータレジスタ番号(000〜111)

◉アドレッシング・モード

sou	size	dest Dn #	Dn ~	Dn #	Dn ~
Dn	B				
	W	2	<40	2	10
	L				
(An)	B				
	W	2	<44	2	14
	L				
(An)+	B				
	W	2	<44	2	14
	L				
−(An)	B				
	W	2	<46	2	16
	L				
d16(An)	B				
	W	4	<48	4	18
	L				
d8(An,IX)	B				
	W	4	<50	4	20
	L				
Abs.W	B				
	W	4	<48	4	18
	L				
Abs.L	B				
	W	6	<52	6	22
	L				
d16(PC)	B				
	W	4	<48	4	18
	L				
d8(PC,IX)	B				
	W	4	<50	4	20
	L				
#Imm	B				
	W	4	<44	4	14
	L				

左側（1番目のDn）：トラップ発生
右側（2番目のDn）：トラップが発生しなかったとき

◉サンプル・リスト

```
CHK     DØ,D3
CHK     (A2),D7
```

111 ●RTE［特権命令］［Return from Exception 例外処理からのリターン］

RTE	X	N	Z	V	C
	*	*	*	*	*

解説　例外処理からの復帰命令で，例外処理の入り口でスタックへ退避してあったシステム情報であるSRとPCを取り出し，例外発生前の状態へ戻します．つまり，取り出したPCの示すアドレスから処理を続行します．

命令実行前のSRとPCは失われ，SRの全ビットが影響を受けます．

CCR

T：スタック上のワードに対応するビット15（トレース）の値

S：スタック上のワードに対応するビット13（スーパバイザ状態）の値

I_2：スタック上のワードに対応するビット10（割り込みマスク）の値'

I_1：スタック上のワードに対応するビット9（割り込みマスク）の値

I_0：スタック上のワードに対応するビット8（割り込みマスク）の値

X：スタック上のワードに対応するビット4の値

N：スタック上のワードに対応するビット3の値

Z：スタック上のワードに対応するビット2の値

V：スタック上のワードに対応するビット1の値

C：スタック上のワードに対応するビット0の値

◉アドレッシング・モード

sou	size	dest	
		＃	～
		2	20

サイズ，ソース／ディスティネーションという概念は,まったく存在しない．

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	0	1	1

◉サンプル・リスト

```
RTE
```

第3部　命令の詳細

112 ◉RESET [特権命令] [Reset external devices 外部デバイス・リセット]

RESET	X	N	Z	V	C
	—	—	—	—	—

解説

MPUのリセット・ラインをアサート(アクティブ)します.

本命令により，プロセッサは124クロック・パルスの間リセット端子を駆動しますが，プロセッサの内部状態に影響を与えることはなく，内部レジスタ，SR，も変更されませんが，PCは本命令の次の命令から継続して命令を実行するため，通常の命令を実行するときのように変更されます.

CCR

X：変化せず
N：変化せず
Z：変化せず
V：変化せず
C：変化せず

◉アドレッシング・モード

sou	size	dest	
		#	~
		2	132

サイズ，ソース／ディスティネーションという概念は，まったく存在しない.

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	0	0	0

◉サンプル・リスト

```
RESET
```

113 ◉STOP［特権命令］［Load Status Register and Stop ステータスレジスタのロードとストップ］

STOP　#〈data〉	X	N	Z	V	C
	*	*	*	*	*

解説

プロセッサの実行を停止するための命令で，次のように動作します．

① 指定したイミディエイト値をSR全体に転送する．
ここでは，割り込みの優先度やトレース状態の設定をするが，Sビット（スーパーバイザ状態）が0なら，特権違反となる．

② PCは更新されて次の命令をポイントするが，プロセッサは命令の取り込み（フェッチ）及び実行を停止する．
ただし，STOP命令を実行する際にSRのT(トレース)ビットがセットされていれば，トレース例外処理を実行する．

再起動されるケースは次の2つです．

① 現在のプロセッサより高い優先度の割り込み要求が発生すると，割り込み例外が発生し，割り込み処理ルーチンを起動できるが，割り込みの優先度がプロセッサと同じか，低い優先度の割り込み要求は無視され，停止状態を保持する．

② 外部リセット

CCR

T：イミディエイト・オペランドに対応するビット15（トレース）の値
S：イミディエイト・オペランドに対応するビット13（スーパバイザ状態）の値
I_2：イミディエイト・オペランドに対応するビット10（割り込みマスク）の値
I_1：イミディエイト・オペランドに対応するビット9（割り込みマスク）の値
I_0：イミディエイト・オペランドに対応するビット8（割り込みマスク）の値
X：イミディエイト・オペランドに対応するビット4の値
N：イミディエイト・オペランドに対応するビット3の値
Z：イミディエイト・オペランドに対応するビット2の値
V：イミディエイト・オペランドに対応するビット1の値
C：イミディエイト・オペランドに対応するビット0の値

◉アドレッシング・モード

sou	size	dest	
		#Imm	
		#	～
		4	4

サイズは存在せず，オペランドはディスティネーションだけが存在する．

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	0	1	0

〈第2ワード〉

15　　　　　　　　　　　　　　0
イミーディエイト・データ

→ 指定したイミディエイト値が格納されるフィールドで，この値は，そのままSRへ代入される内容でもある．

◉サンプル・リスト

```
STOP    #MASK
```
←MASKとはプログラマが定義した定数である

114 ◉NOP [No Operation ノー・オペレーション]

NOP	X	N	Z	V	C
	—	—	—	—	—

解説

何も実行しません.

PCは本命令の次に位置する命令から継続して実行するために更新されますが，その他は，プロセッサの状態に何の影響も与えません.

CCR

X：変化せず
N：変化せず
Z：変化せず
V：変化せず
C：変化せず

◉アドレッシング・モード

sou	size	dest	
		#	～
		2	4

サイズ，ソース，ディスティネーションという概念は存在しない.

◉機械語フォーマット

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	0	0	1

◉サンプル・リスト

```
NOP
```

索引

【参考文献】

M68000マイクロプロセッサ・ユーザーズ・マニュアル　第4版：CQ出版社
16ビット・マイクロコンピュータとプログラミングの基礎——MC68000のアセンブラ入門——福永邦雄著：CQ出版社
インターフェース　1981/12 No.55　特集/16ビット・マイコン・システム設計の基礎：CQ出版社
68000マイクロコンピュータ：森亮一監修：丸善
68000プログラミング入門：Tim King, Brian Knight著．鈴木隆監訳：アスキー
日立マイクロコンピュータ・データブック　8/16ビットマイクロプロセッサ：日立製作所
16ビットトレーニングモジュールH680TR01ユーザーズマニュアル：日立製作所
68000Assembly Language Programming 1981年版．by Gerry Kane, Doug Hawkins and Lance Leventhal：Osborne／McGraw-Hill
MC68000 16-BIT MICROPROCESSOR User's Manual Second Edition (MOTOROLA)
MC68010 16-BIT VIRTUAL MEMORY MICROPROCESSOR (DECEMBER, 1982) (MOTOROLA)
MC68230 PARALLEL INTERFACE/TIMER(PI/T) (DECEMBER, 1983) (MOTOROLA)
MC68901 MULTI-FUNCTION PERIPHERAL (JANUARY, 1984) (MOTOROLA)
NEC　PC-9800シリーズ　68000ボード・サポートソフトウェア・ユーザーズ・マニュアル(NEC)
M68000 Family Resident Structured Assembler Manual (MOTOROLA)
16 Bit Microprocessor Programming Card (MOTOROLA)

68000プログラマーズ・ハンドブック

昭和61年9月20日　初版　第1刷発行
昭和63年1月10日　初版　第4刷発行

著　者　宍倉幸則（ししくら ゆきのり）
発行者　片岡　巌
発行所　株式会社技術評論社
東京都千代田区九段南2-4-13
電話　03(262)9351　代表・営業部
03(262)7671　代表・編集部
印　刷
製　本　図書印刷

定価はカバーに表示してあります．

ISBN4-87408-838-4 C3055

●実行命令オペランド一覧表

＊裏面に続く

命令	サイズ			ソース・オペランド												ディスティネーション・オペランド												コンディション・コード					備考	参照ページ
	B	W	L	Dn	An	(An)	(An)+	−(An)	d16(An)	d8(An, IX)	Abs	d16(PC)	d8(PC, IX)	#Imm	SR/CCR	Dn	An	(An)	(An)+	−(An)	d16(An)	d8(An, IX)	Abs	d16(PC)	d8(PC, IX)	#Imm	SR/CCR	X	N	Z	V	C		
MOVE	●	●	●	●	●	●	●	●	●	●	●	●	●	●		●		●	●	●	●	●	●					—	*	*	0	0		64, 266, 269, 270, 271, 272, 273
MOVE		●		●		●	●	●	●	●	●	●	●	●														CCR	*	*	*	*	*	下位8ビットだけが対象	
MOVE		●		●		●	●	●	●	●	●	●	●	●														SR	*	*	*	*	*	特権命令	
MOVE		●														SR	●		●	●	●	●	●	●					—	—	—	—	—		
MOVE			●		USP													●											—	—	—	—	—	特権命令	
MOVE			●		●													USP											—	—	—	—	—	特権命令	
MOVEA		●	●	●	●	●	●	●	●	●	●	●	●	●			●											—	—	—	—	—		65, 268
MOVEM		●	●	レジスタ・リスト															●		●	●	●	●					—	—	—	—	—		65, 274, 276
MOVEM		●	●			●	●		●	●	●	●	●			レジスタ・リスト													—	—	—	—	—		
MOVEP		●	●	●																		●							—	—	—	—	—		65, 278, 280
MOVEP		●	●						●							●												—	—	—	—	—		
MOVEQ			●											●		●												—	*	*	0	0		65, 282
EXG			●	●												●	●											—	—	—	—	—		66, 283
EXG			●		●											●	●											—	—	—	—	—		
LEA			●			●			●	●	●	●	●				●											—	—	—	—	—		66, 285
SWAP		●															●												—	*	*	0	0		66, 284
PEA			●			●			●	●	●	●	●																—	—	—	—	—		66, 286
LINK	—	—	—		●																					●		—	—	—	—	—		66, 287
UNLK	—	—	—		●																							—	—	—	—	—		66, 288
ADD	●	●	●	●	●	●	●	●	●	●	●	●	●	●		●												*	*	*	*	*		69, 289, 290
ADD	●	●	●	●																●	●	●	●	●	●					*	*	*	*	*		
ADDA		●	●	●	●	●	●	●	●	●	●	●	●	●			●											—	—	—	—	—		69, 291
ADDI	●	●	●											●		●		●	●	●	●	●	●					*	*	*	*	*		69, 292
ADDQ	●	●	●											●		●	●(注)	●	●	●	●	●	●					*	*	*	*	*	(注)バイト・サイズ不可	69, 294
ADDX	●	●	●	●												●												*	*	*	*	*		295, 296
ADDX	●	●	●					●													●							*	*	*	*	*		
SUB	●	●	●	●	●(注)	●	●	●	●	●	●	●	●	●		●												*	*	*	*	*	(注)バイト・サイズ不可	297, 298
SUB	●	●	●	●																●	●	●	●	●	●					*	*	*	*	*		
SUBA		●	●	●	●	●	●	●	●	●	●	●	●	●			●											—	—	—	—	—		70, 299
SUBI	●	●	●											●		●		●	●	●	●	●	●					*	*	*	*	*		70, 300
SUBQ	●	●	●											●		●	●(注)	●	●	●	●	●	●					*	*	*	*	*	(注)バイト・サイズ不可	70, 301
SUBX	●	●	●	●												●												*	*	*	*	*		70, 302, 303
SUBX	●	●	●					●													●							*	*	*	*	*		
MULS		●		●		●	●	●	●	●	●	●	●	●		●												—	*	*	0	0		70, 304
MULU		●		●		●	●	●	●	●	●	●	●	●		●												—	*	*	0	0		70, 305
DIVS		●		●		●	●	●	●	●	●	●	●	●		●												—	*	*	*	0		70, 306
DIVU		●		●		●	●	●	●	●	●	●	●	●		●												—	*	*	*	0		70, 308
CLR	●	●	●													●		●	●	●	●	●	●					—	0	1	0	0		71, 314
NEG	●	●	●													●		●	●	●	●	●	●					*	*	*	*	*		71, 316
NEGX	●	●	●													●		●	●	●	●	●	●					*	*	*	*	*		71, 317
TAS	●															●		●	●	●	●	●	●					—	*	*	0	0		71, 319
TST	●	●	●													●		●	●	●	●	●	●					—	*	*	0	0		71, 318
EXT		●	●													●												—	*	*	0	0		71, 315
CMP	●	●	●	●	●(注)	●	●	●	●	●	●	●	●	●		●												—	*	*	*	*	(注)バイト・サイズ不可	71, 310
CMPA		●	●	●	●	●	●	●	●	●	●	●	●	●			●											—	*	*	*	*		71, 311
CMPI	●	●	●											●		●		●	●	●	●	●	●					—	*	*	*	*		71, 312
CMPM	●	●	●				●												●								—	*	*	*	*		71, 313
AND	●	●	●	●		●	●	●	●	●	●	●	●	●		●												—	*	*	0	0		72, 320, 321
AND	●	●	●	●																●	●	●	●	●	●					—	*	*	0	0		
ANDI	●	●	●											●		●		●	●	●	●	●	●					—	*	*	0	0		73, 322, 323, 324
ANDI	●	●												●														●	*	*	*	*	*	SR形式は特権命令	
OR	●	●	●	●		●	●	●	●	●	●	●	●	●		●												—	*	*	0	0		73, 329, 330
OR	●	●	●	●																●	●	●	●	●	●					—	*	*	0	0		
ORI	●	●	●											●		●		●	●	●	●	●	●					—	*	*	0	0		73, 331, 332, 333
ORI	●	●												●														●	*	*	*	*	*	SR形式は特権命令	
EOR	●	●	●	●												●		●	●	●	●	●	●					—	*	*	0	0		73, 325
EORI	●	●	●											●		●		●	●	●	●	●	●					—	*	*	0	0		74, 326, 327, 328
EORI	●	●												●														●	*	*	*	*	*	SR形式は特権命令	
NOT	●	●	●													●		●	●	●	●	●	●					—	*	*	0	0		74, 334
ASL	●	●	●	●												●												*	*	*	*	*		75, 76, 335, 336, 337
ASL	●	●	●											●		●												*	*	*	*	*		
ASL		●																	●	●	●	●	●	●					*	*	*	*	*		
ASR	●	●	●	●												●												*	*	*	0	*		76, 338, 339, 340
ASR	●	●	●											●		●												*	*	*	0	*		
ASR		●																	●	●	●	●	●	●					*	*	*	0	*		

命令	サイズ			ソース・オペランド												ディスティネーション・オペランド												コンディション・コード					備考	参照ページ
	B	W	L	Dn	An	(An)	(An)+	-(An)	d16(An)	d8(An, IX)	Abs	d16(PC)	d8(PC, IX)	#Imm	SR/CCR	Dn	An	(An)	(An)+	-(An)	d16(An)	d8(An, IX)	Abs	d16(PC)	d8(PC, IX)	#Imm	SR/CCR	X	N	Z	V	C		
LSL	●	●	●	●												●												*	*	*	0	*		76, 341, 342, 343
LSL	●	●	●											●		●												*	*	*	0	*		
LSL		●																●	●	●	●	●	●					*	*	*	0	*		
LSR	●	●	●	●												●												*	*	*	0	*		76, 344, 345, 346
LSR	●	●	●											●		●												*	*	*	0	*		
LSR		●																●	●	●	●	●	●					*	*	*	0	*		
ROL	●	●	●	●												●												—	*	*	0	*		76, 347, 348, 349
ROL	●	●	●											●		●												—	*	*	0	*		
ROL		●																●	●	●	●	●	●					—	*	*	0	*		
ROR	●	●	●	●												●												—	*	*	0	*		351, 352
ROR	●	●	●											●		●												—	*	*	0	*		
ROR		●																●	●	●	●	●	●					—	*	*	0	*		
ROXL	●	●	●	●												●												*	*	*	0	*		76, 353, 354, 355
ROXL	●	●	●											●		●												*	*	*	0	*		
ROXL		●																●	●	●	●	●	●					*	*	*	0	*		
ROXR	●	●	●	●												●												*	*	*	0	*		76, 356, 357, 358
ROXR	●	●	●											●		●												*	*	*	0	*		
ROXR		●																●	●	●	●	●	●					*	*	*	0	*		
BCHG	●		●	●												●		●	●	●	●	●	●					—	—	*	—	—	ディスティネーション・オペランドがメモリ・オペランドなら，サイズはバイトに限定される	78, 365, 366
BCHG	●		●											●		●		●	●	●	●	●	●					—	—	*	—	—		
BCLR	●		●	●												●		●	●	●	●	●	●					—	—	*	—	—	ディスティネーション・オペランドがメモリ・オペランドなら，サイズはバイトに限定される	78, 363, 364
BCLR	●		●											●		●		●	●	●	●	●	●					—	—	*	—	—		
BSET	●		●	●												●		●	●	●	●	●	●					—	—	*	—	—	ディスティネーション・オペランドがメモリ・オペランドなら，サイズはバイトに限定される	78, 361, 362
BSET	●		●											●		●		●	●	●	●	●	●					—	—	*	—	—		
BTST	●		●	●												●		●	●	●	●	●	●	●	●		CCR	—	—	*	—	—	ディスティネーション・オペランドがメモリ・オペランドなら，サイズはバイトに限定される	78, 359, 360
BTST	●		●											●		●		●	●	●	●	●	●	●	●			—	—	*	—	—		
ABCD	●			●												●												*	U	*	U	*		79, 367, 368
ABCD	●							●												●								*	U	*	U	*		
SBCD	●			●												●												*	U	*	U	*		79, 369, 370
SBCD	●							●												●								*	U	*	U	*		
NBCD	●															●		●	●	●	●	●	●					*	U	*	U	*		79, 371
Bcc	●	●																										—	—	—	—	—		81, 372
DBcc		●		●																								—	—	—	—	—		81, 374
Scc	●															●		●	●	●	●	●	●					—	—	—	—	—		82, 376
BRA	●	●																										—	—	—	—	—		82, 378
BSR	●	●																										—	—	—	—	—		82, 379
JMP	—	—	—															●			●	●	●	●	●			—	—	—	—	—		82, 380
JSR	—	—	—															●			●	●	●	●	●			—	—	—	—	—		82, 381
RTE	—	—	—																									*	*	*	*	*	特権命令	87, 387
RTR	—	—	—																									*	*	*	*	*		82, 382
RTS	—	—	—																									—	—	—	—	—		82, 383
RESET	—	—	—																									—	—	—	—	—	特権命令	87, 388
STOP	—	—	—											●														*	*	*	*	*	特権命令	87, 389
CHK		●		●		●	●	●	●	●	●	●	●	●		●												—	*	U	U	U		85, 386
TRAP	—	—	—											●														—	—	—	—	—		85, 384
TRAPV	—	—	—																									—	—	—	—	—		85, 385
NOP	—	—	—																									—	—	—	—	—		88, 390

【68000 比較命令と条件】

CMP X, Y

○内部演算は (Y)−(X) を行う。
○「(Y)は(X)より大きい」などと表現されるので，主語は第２オペランドとなる。

条件式	符号なし	符号あり
Y ＞ X	HI	GT
Y ≧ X	CC	GE
Y ＝ X	EQ	EQ
Y ≠ X	NE	NE
Y ≦ X	LS	LE
Y ＜ X	CS	LT

68000プログラマーズ・ハンドブック

宍倉幸則 著

技術評論社

ISBN4-87408-838-4 C3055 ¥2900E　定価 2,900円